

RT-Thread实时操作系统编程指南

版本 ***0.3.0***

RT-Thread工作室

March 18, 2010

CONTENTS

Contents:

CHAPTER

ONE

序

1.1 RT-Thread诞生

RT-Thread 实时操作系统，Kernel部分完成于2006年上半年，其IPC部分甚至是年中时才具备相应的雏形。最开始时是因为要为朋友做一个小型的手持设备，而本人起初又是另一国内老牌实时操作系统：DOOLOO RTOS开发人员，但这个团队在2005年底已经解散。但朋友的系统要上，用其他小型系统吗，一不熟悉，二看不上。答应朋友的事，总得有解决方法吧，即使是原来的DOOLOO RTOS，因为其仿VxWorks结构，导致它的核心太大，包括太多不必要的东西（一套完整的libc库），这些方案都否决了。怎么办？当时朋友那边也不算太急，先自己写一套内核吧。这个就是源头！（后来虽然朋友的项目夭折了，但这套OS则保留下来了，并开源了，万幸）

当然RT-Thread和原来的DOOLOO RTOS差别还是很大的。DOOLOO RTOS是一种类VxWorks风格的，而RT-Thread则是一种追求小型风格的实时操作系统：小型、实时、可剪裁。这三个方面RT-Thread可以骄傲的说做得比DOOLOO RTOS都要好很多，小型：RT-Thread核心能够小到2.5K ROM，1K RAM；实时：线程调度核心是完全bitmap方式，计算时间是完全固定的；可剪裁性，配置文件rtconfig.h包含多种选项，对Kernel细节进行精细调整，对各种组件（文件系统，使用EFSL、ELM FatFs；网络协议栈，finsh shell，图形用户界面GUI）进行可选配置。

1.2 艰难的发展期

在第一个公开板发布后（0.1），RT-Thread意识到了一个问题，光有核心不行。别人如何使用：虽然采用了doxygen风格的注释，并自动产生相应的API文档(且是英文的)，但能够使用的人寥寥，有这个功底的人不见得认可你的系统，没相应功底的人也玩不转你的系统。所以下一个系列，考虑如何让系统能够支持更多的平台。首选ARM，为什么？因为ARM正处于发展的前期，使用的人也广泛，而RT-Thread第一个支持的平台就是s3c4510，这个是 lumit 开源项目赠送的平台。在其后，支持了包括 s3c44b0，AT91SAM7S64，AT91SAM7X256，s3c2410，AT91SAM9200，coldfire，x86等一系列平台，编译器统一使用GCC，这个时期无疑是最艰难的时期(真的艰难吗？或许不是，但肯定是迷茫的)，这期间陆续发布了0.2.0、0.2.1、0.2.3、0.2.4版本等，不同的版本支持不同的平台。

在这个苦中做乐的日子里，shaolin同学出现了，帮助完成了 RT-Thread/x86的移植，他当时还是学生，同时也把RT-Thread作为了他的毕业设计论文。

1.3 一年增加0.0.1

本人很早就接触了Linux，算是国内资深的Linux接触者（早期也算一个Linux开发人员吧），KDE 1.0几乎是看着发展起来的（大家有谁用过RedHat 5.1？）。个人算是很多方面有一些自由软件的习惯：软件的版本号是非常重要的一个标志，宁愿增加一个细微的版本号也不轻易的增加一个大的版本号，因为大的版本号是需要对用户负责的。1.0版本更代表了系统的稳定性，健全性。例如mplayer到1.0版本就经历众多小版本，0.99的beta版本亦无数。

RT-Thread也把这点体现得淋漓尽致，0.2.2到0.2.3一个版本的增加，整整花了一年多的时间。但这个小版本号的增加，却带来了开源社区嵌入式环境中最完善的TCP/IP协议栈：LwIP。当然，开始时并不算稳定。在这几个版本中，RT-Thread也终于从迷茫中走出来，RT-Thread需要自己的特色，一个单独的实时操作系统内核没太大的用处，因为你并没有上层应用代码的积累支撑，并且一些基础组件也非常重要，有这些基础组件基本上意味着，在这个平台上写代码，这些代码就属于你自己的，甚至于哪天也可以把它放到另外一个硬件平台上运行。

同样，0.2到0.3版本号的变更，花费的时间会更长：版本号的长短，是和计划的功能特性密切相关的，没到设定的目标如何可能进行发布？

1.4 Cortex-M3的变革

RT-Thread的变革因为 Cortex-M3 而来，因为ST的 STM32 使用的人太广了，当然还有非常重要的一点。RT-Thread已经开始支持 Keil MDK ，armcc编译器了。 GNU GCC 确实好，并且也由衷的推崇它，使用它，只是调试确实麻烦，阻碍了更多人使用它（ARM平台上）。当 RT-Thread + Cortex-M3 + Keil MDK碰撞在一起的时候，火花因它而生，越来越多人使用RT-Thread了，当然这和RT-Thread厚积薄发是离不开的，因为这个时候，RT-Thread已经有一个稳定的内核，shell方式的调试利器finsh，DFS虚拟设备文件系统，LwIP协议栈以及自己从头开发的图形用户界面GUI。RT-Thread/GUI成型于2008年底，但为了 Cortex-M3分支，这个组件停下来很多，但这种停留是值得的。另外就是特别感谢UET赠送的STM32开发板了，RT-Thread/STM32的分支都是在UET赠送的STM32开发板上验证的。

在Cortex-M3这个平台上(当然也包括其他一些ARM平台)，已经有多家企业开始使用RT-Thread实时操作系统作为他们的软件平台，其中也包括一家国内A股上市企业，他们把RT-Thread应用于电力，监控，RFID，数据采集等数个领域中，RT-Thread实时操作系统涵盖了实时内核，文件系统，LwIP网络协议栈，finsh shell等组件。RT-Thread/GUI因为出道时间比较晚，还处于试用期:-)

1.5 面向对象设计方法

了解RT-Thread开源实时操作系统的都知道，采用面向对象风格的设计是RT-Thread一个很大的特点，但它又不等同于eCos操作系统那样纯粹使用C++来实现，而是采用了一种C编码的面向对象编程。面向对象设计更适合于人类思考问题的特点，有着它天然的好处，例如继承。可以让具备相同父类的子类共享使用父类的方法，基本可以说是不用写代码就凭空多出了很多函数，何乐而不为。另外，对象的好处在于封装。当一个对象封装好了以后，并测试完成后，基本上就代表这个类是健全的，从这个类派生的子类不需要过多考虑父类的不稳定性。

这里着重提提另外一个人，我工作后的第三年，曾向当时的同事也是好友，L.Huray学习面向对象的实时设计方法：Octpus II。深刻体会到了面向对象设计的好处（需求分析，体系结构设计，子系统分析，子系统设计，子系统测试，集成测试，实时性分析等），但鉴于嵌入式系统中C++的不确定性，所以个人更偏向于使用C语言来实现。所以，L.Huray算是我的老师了，一直希望能够有

时间把他老人家的思想更进一步的发扬光大，希望以后有这个机会。（Octpus I最初起源于Nokia，然后由M.Award, L.Huray发展成Octpus II，现在几乎见不到踪影了）。

1.6 文档结构

本书是RT-Thread实时操作系统的编程指南文档，它分几个部分分别描述了:

- 实时系统概念：实时系统是一个什么样的系统，它的特点是什么；
- RT-Thread快速入门，在无硬件平台的情况下，如何迅速地了解RT-Thread实时操作系统，如何使用RT-Thread实时操作系统最基本的一些元素；
- RT-Thread作为一个完整的实时操作系统，它能够满足各种实时系统的需求，所以接下来详细地介绍了各个模块的结构，以及编程时的注意事项。
- RT-Thread外围组件的编程说明，RT-Thread不仅包括了一个强实时的内核，也包括外围的一些组件，例如shell，文件系统，协议栈，图形用户界面等。这部分对外围组件编程进行了描述。
- RT-Thread中一些其他部分说明，包含了如何使用GNU GCC工具搭建RT-Thread的开发环境及RT-Thread在Cortex-M3系统上的说明。

本书面向使用RT-Thread系统进行编程的开发人员，并假定开发人员具备基本的C语言基础知识，如果具备基本的实时操作系统知识将能够更好地理解书中的一些概念。本书是一本使用RT-Thread进行编程的书籍，对于RT-Thread的内部实现并不做过多、过细节性的分析。

本书中异常与中断由王刚编写，定时器与系统时钟，I/O设备管理，文件系统由邱祎编写，其余部分由熊谱翔编写，部分章节由李进进行审校。

熊谱翔于上海 March 18, 2010

CHAPTER

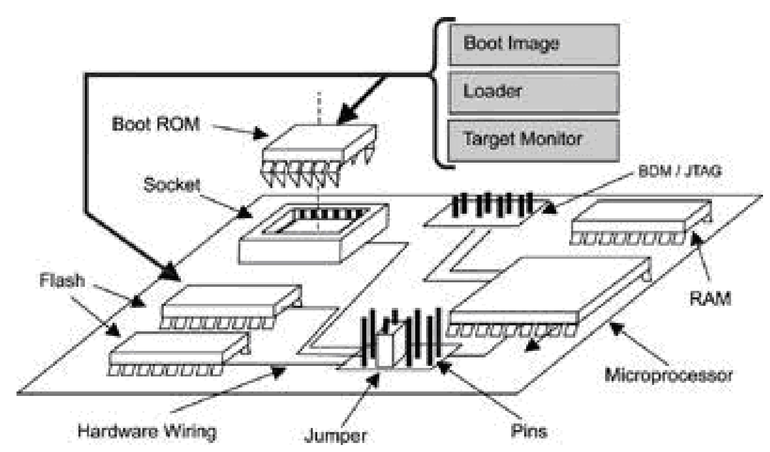
TWO

实时系统

2.1 嵌入式系统

嵌入式系统是为满足特定需求而设计的计算系统，常见的嵌入式系统如：电视用的机顶盒，网络中的路由器等。它们总是针对特定的需求，例如电视机顶盒，用于播放网络中的电视节目(不会试图用来写文档)；网络路由器，用于网络报文的正确转发(不会试图用于游戏，看电影)。这类系统通常针对特定的外部输入进行处理然后给出相应的结果。功能相对单一(因为需求也相对单一)，而正是因为这类系统的专用性，为了完成这一功能，嵌入式系统提供相匹配的硬件资源，多余的硬件资源能力是浪费，而欠缺的硬件资源能力则不能够满足设定的目标，即在成本上“恰好”满足设定的要求。

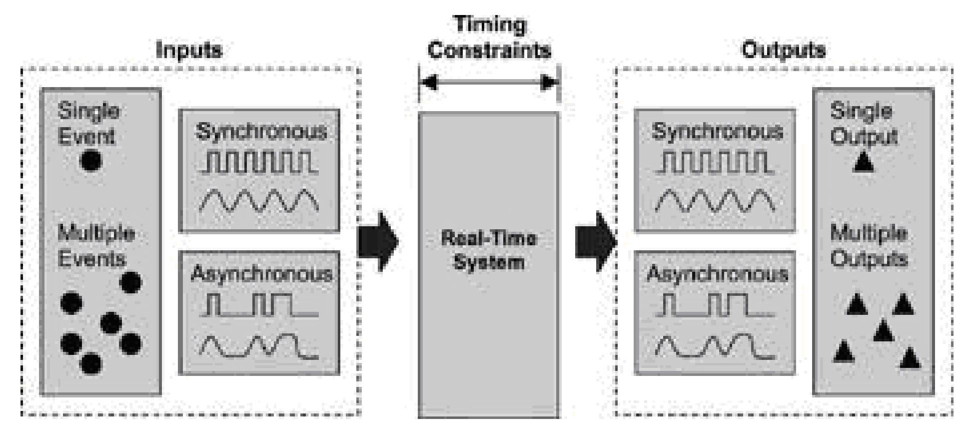
通用系统则恰恰相反，他们并不具备特定的需求，是力图尽可能得满足最大化的需求，甚至在构造硬件系统时还会考虑到满足未来几年的需求变化。例如，在人们购买电脑时，在自身有限的资金情况下，尽可能获得高端的性能，用于多媒体，游戏，工作等。



嵌入式系统的硬件设备由一些芯片及电路组成，微控制器，存放代码的Flash，Boot Rom，运行代码时用到的内存(开发阶段调试时用到的JTAG接口)等，也可能包含一定的机械控制模块，数据采集模块等，在控制芯片当中会包含相应的计算单元。总的来说，嵌入式系统提倡的是在满足设定需求的条件下，力图以最小的的成本代价完成相应的功能。嵌入式系统通常会进行大量生产。所以单个的成本节约，能够随着产量进行成百上千的放大。

2.2 实时系统

实时计算可以定义成这样一类计算，即系统的正确性不仅取决于计算的逻辑结果，而且还依赖于产生结果的时间。关键的两点：正确地完成和在给定的时间内完成，且两者重要性是等同的。针对于在给定的时间内功能性的要求可以划分出常说的两类实时系统，软实时和硬实时系统。如下一个示例图：



对于输入的信号、事件，实时系统必须能够在规定的时间内得到正确的响应，而不管这些事件是单一事件、多重事件还是同步信号或异步信号。一个具体的例子：一颗子弹从20米处射出，射向一个玻璃杯。假设子弹的速度是v米/秒，那么经过t1=20/v秒后，子弹将击碎玻璃杯。而有一系统在检测到子弹射出后，将把玻璃杯拿走，假设这整个过程将持续t2秒的事件。如果t2 < t1，那么这个系统可以看成是一个实时系统。

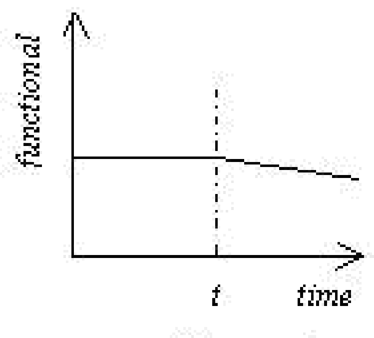
和嵌入式系统类似，实时系统上也存在一定的计算单元，对系统的环境、里面的应用有所预计，也就是很多实时系统所说的确定性：对一个给定事件，在一给定的事件t秒内做出响应。对多个事件、多个输入的响应的确定性构成了整个实时系统的确定性。（实时系统并不代表着对所有输入事件具备实时响应，而是针对指定的事件能够做出实时的响应）

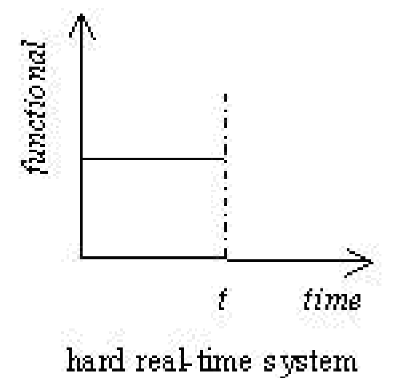
嵌入式系统的应用领域十分广泛，并不是其所针对的专用功能都要求实时性的，只有当系统中对任务有严格时间限定时，才有系统的实时性问题。具体的例子包括实验控制、过程控制设备、机器人、空中交通管制、远程通信、军事指挥与控制系统等。而对打印机这样一个嵌入式应用系统，人们并没有严格的时间限定，只有一个”尽可能快的”期望要求，因此，这样的系统称不上是实时系统。

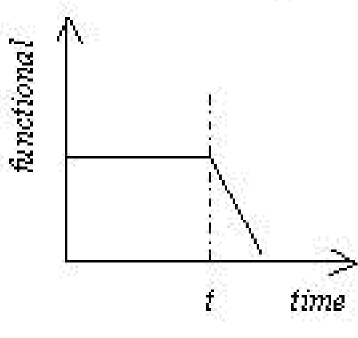
2.3 软实时与硬实时

如上节描述，实时系统非常强调两点：时间和功能的正确性。衡量一个实时系统的正确性也正是这样：在给定的时间内正确地完成相应的任务。但也存在这样一种系统，大部分情况下能够严格的规定的时间内完成任务，但偶尔也会在给定时间之外一点点才能正确地完成任务，这种系统通常称为软实时系统。对规定时间的敏感性构成了硬实时系统和软实时系统的区别。硬实时系统严格限定在规定的时间内完成任务，否则就可能导致灾难的发生，例如导弹的拦截，汽车引擎系统等，当不能满足需求的时间性时，将可能发生车毁人亡的重大灾难，即使是偶尔。而软实时系统，可以允许偶尔出现一定的时间偏差，但是随着时间的偏移，整个系统的正确性也随之下降，例如一个DVD播放系统可以看成一个软实时系统，可以允许它偶尔的画面或声音延迟。

三种系统(非实时系统，硬实时系统和软实时系统)的时效关系可以用下图直观的表示出来：







如上图所示，从功能性、和时间的角度上，三者关系是：

- 非实时系统随着给定时间t的推移，效用缓慢的下降。
- 硬实时系统在给定时间t之后，马上变为零值。
- 软实时系统随着给定时间t的推移，效用迅速的走向零值。

CHAPTER

THREE

快速入门

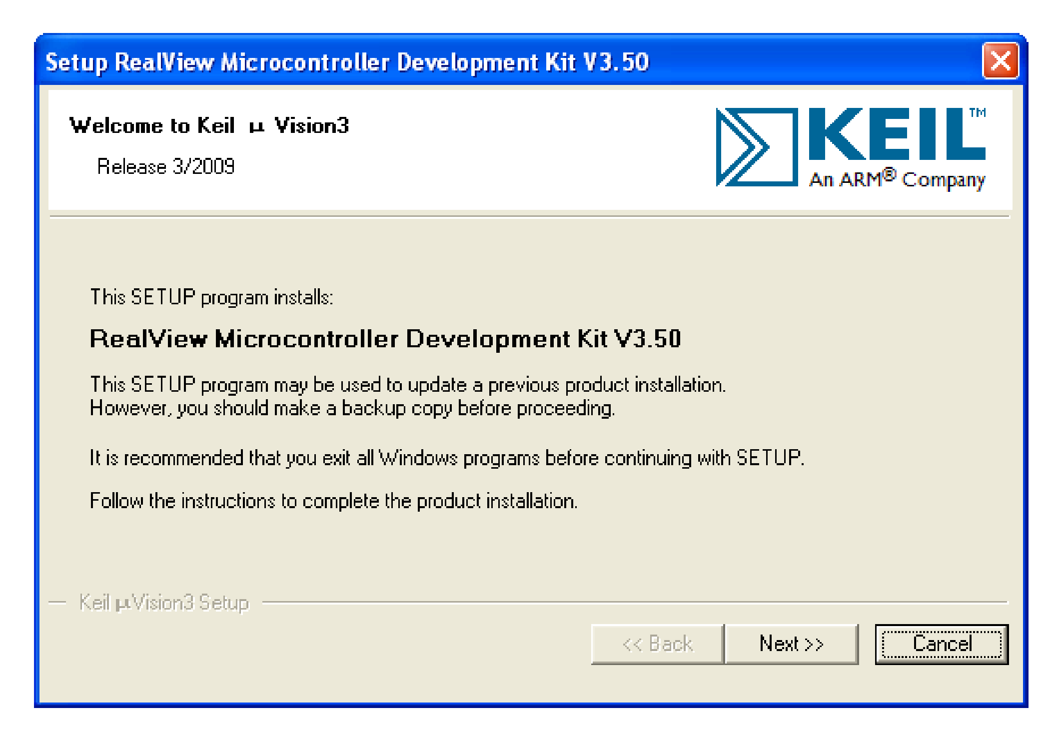
一般嵌入式操作系统因为它的特殊性，往往和硬件平台密切相关连，具体的嵌入式操作系统往往只能在特定的硬件上运行。对于刚接触的读者并不容易马上就获得一个和RT-Thread相配套的硬件模块。在科技发展的今天，还有一种技术叫做仿真运行，下面我们将选择RealView MDK开发环境作为目标平台来看看RT-Thread是如何运行的。RealView MDK开发环境因为其完全的AT91SAM7S64软件仿真环境，让我们有机会在不使用真实硬件平台的情况下运行目标代码。这个软件仿真能够完整地虚拟出ARM7TDMI的各种运行模式，几乎和真实的硬件平台完全一致。实践也证明，这份软件仿真的RT-Thread代码能够在无任何修改的情况下在真实硬件平台中正常运行。

3.1 准备环境

在运行RT-Thread前，我们需要安装RealView MDK(正式版或评估版) 3.5+，它不仅是软件仿真工具，也是编译器链接器。这里采用了16k编译代码限制的评估版3.50版本。

先从Keil官方网站下载 RealView MDK评估版 。

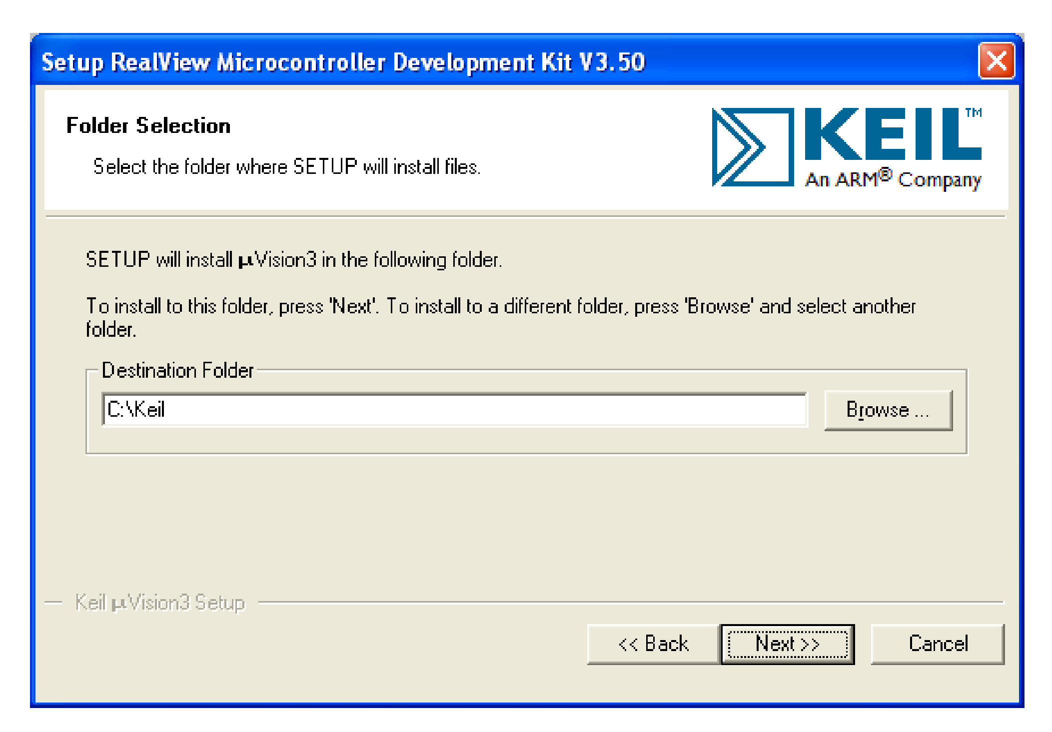
会需要先填一些个人基本信息即可进行下载。下载完毕后，在Windows环境下运行它，会出现如下画面



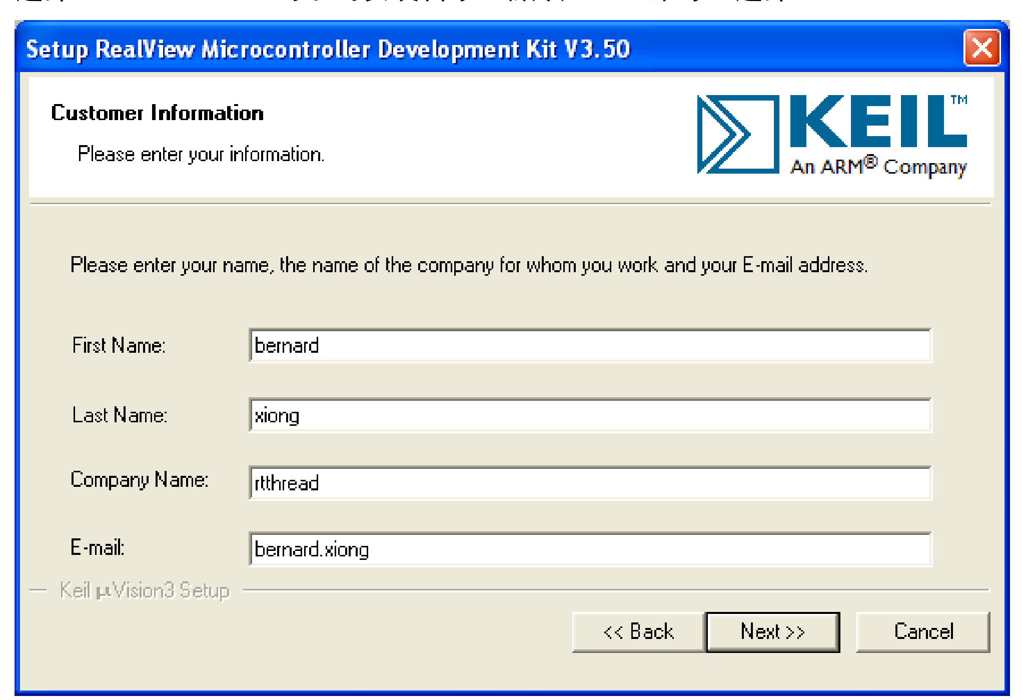
这个是RealView MDK的安装说明，点”Next >>”进入下一画面



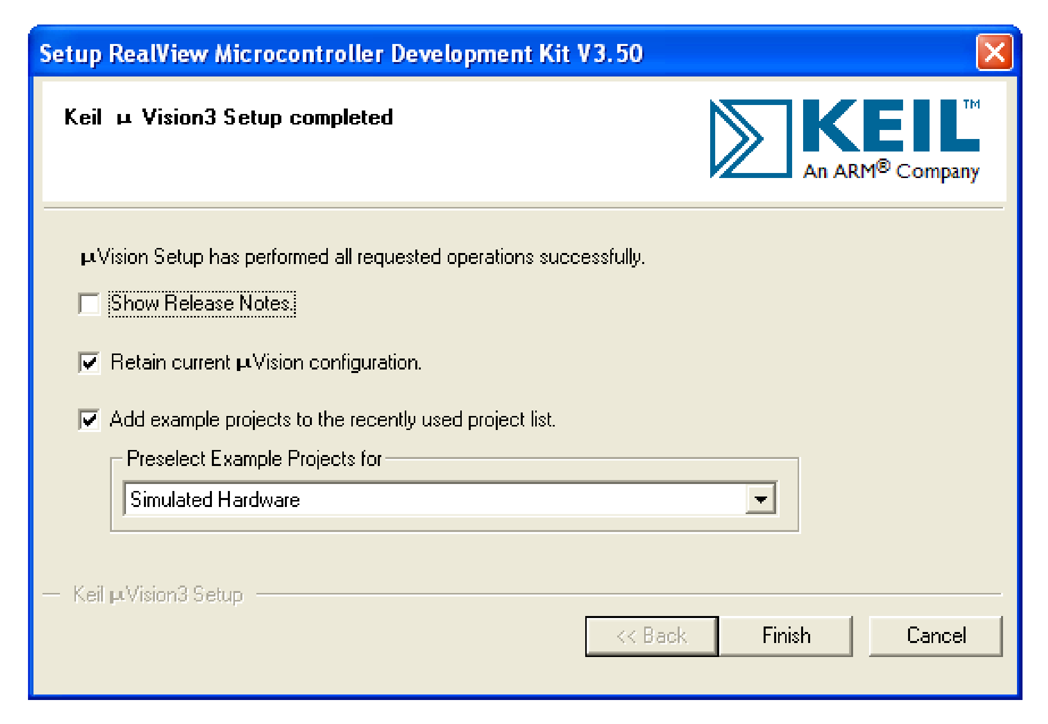
为了能够正常安装，需要同意它的条款(这是一款评估版)，选择”Next >>”



选择RealView MDK对于的安装目录，默认C:Keil即可，选择"Next >>"



输入您的名字及邮件地址，选择"Next >>"



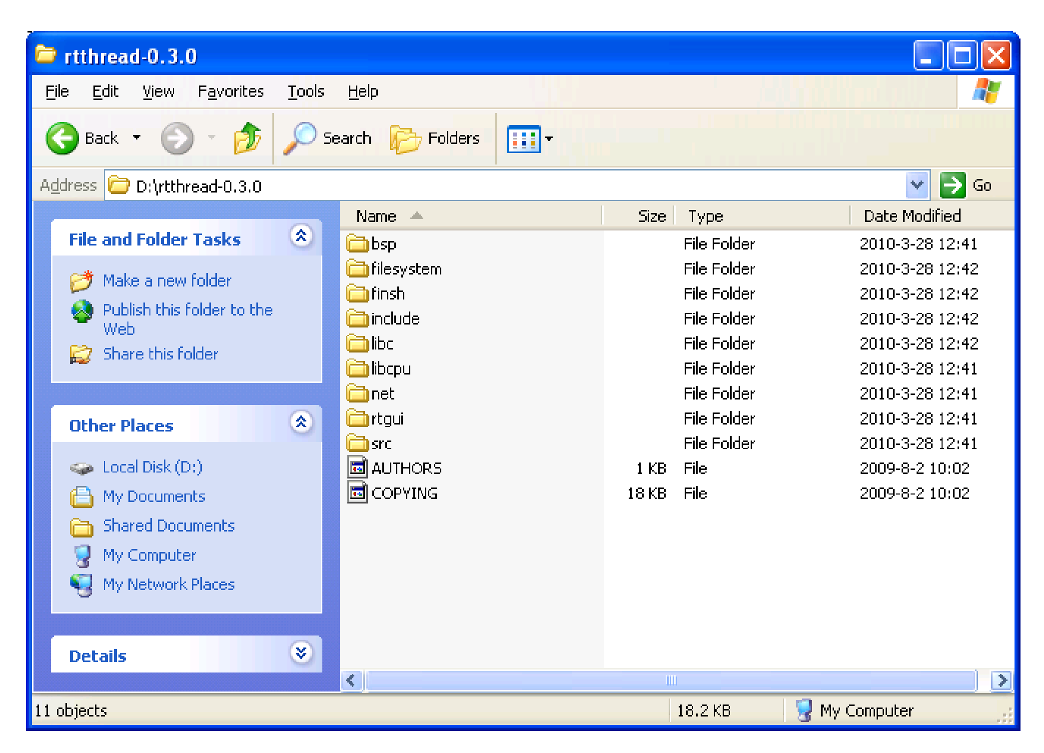
安装完成，选择"Finish"

有了RealView MDK的利器，就可以轻松开始RT-Thread之旅，一起探索实时操作系统的奥秘。请注意RealView MDK正式版是收费的，如果您希望能够编译出更大体积的二进制文件，请购买RealView MDK正式版。RT-Thread也支持自由基金会的GNU GCC编译器，这是一款开源的编译器，想要了解如何使用GNU的相关工具搭建RT-Thread的开发环境请参考本书后面的章节，其中有在Windows/Linux环境下搭建采用GNU GCC做为RT-Thread开发环境的详细说明。

3.2 初识RT-Thread

RT-Thread做为一个操作系统，它的代码规模会不会和Windows或Linux一样很庞大？代码会不会达到惊人的上百万行代码级别？弄清楚这些之前，我们先要做的就是获得本书相对应的RT-Thread 0.3.0正式版 RT-Thread 0.3.0正式版本下载 。

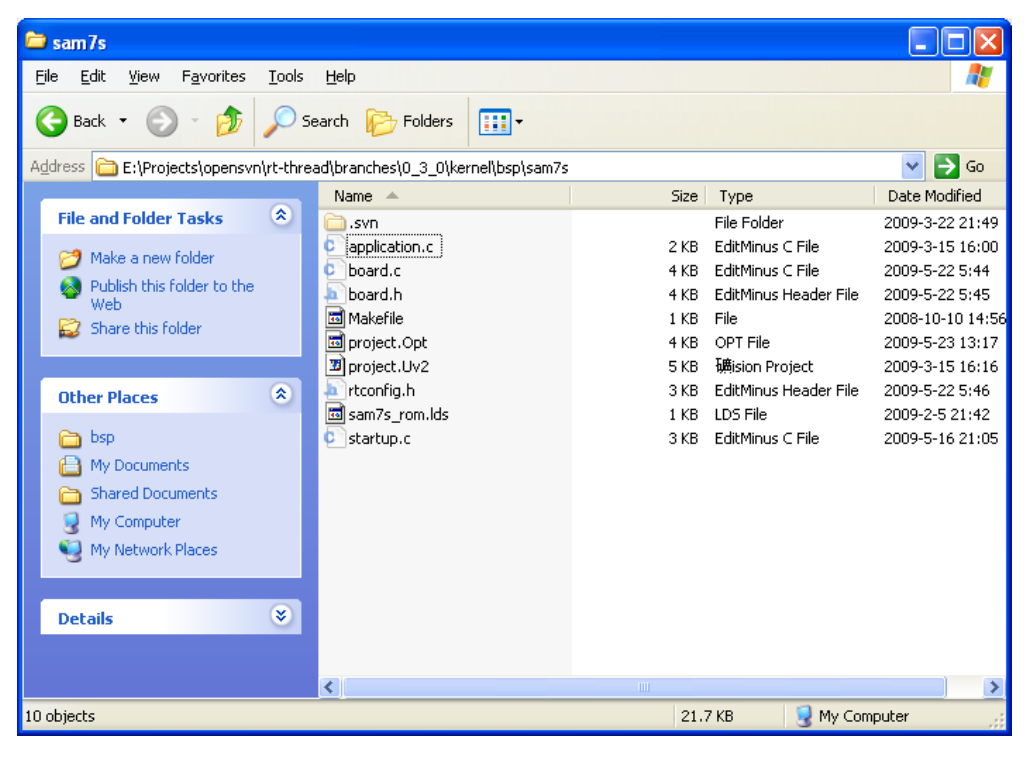
这个是一个压缩包文件，请解压到一个目录，这里把它解压到D:目录中。解压完成后的目录结构是这样的：



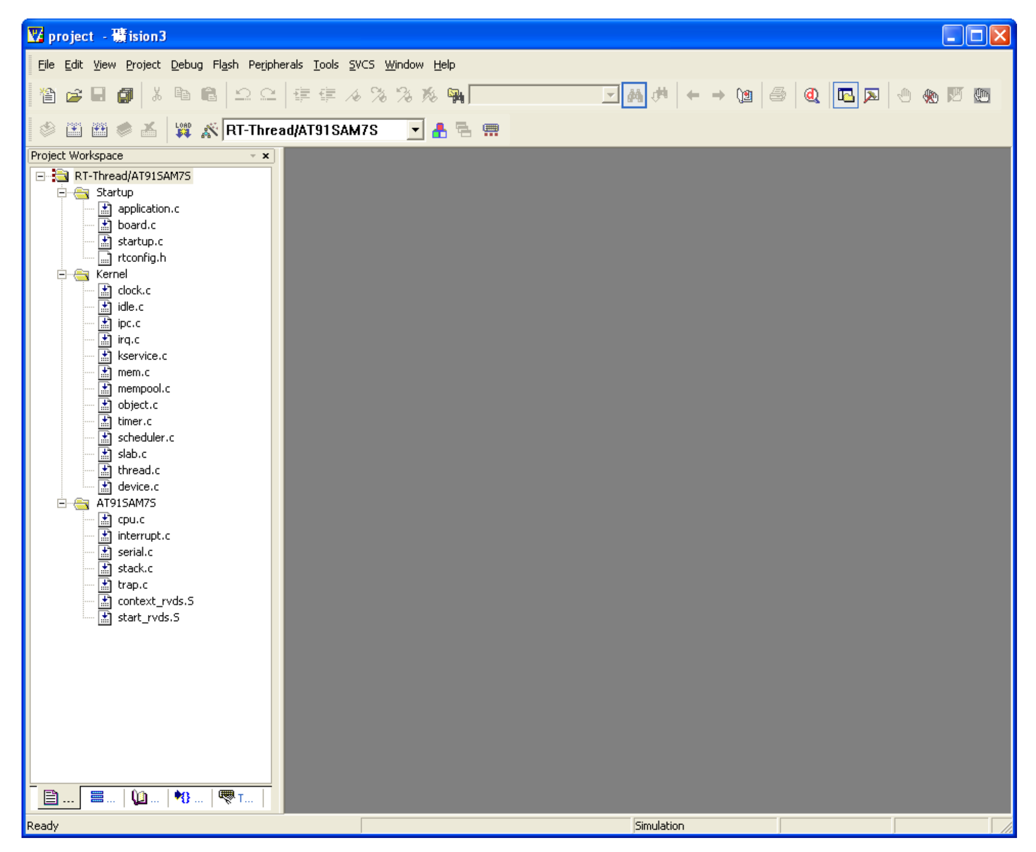
各个目录的解释如下：(请注意在RT-Thread 0.3.0中，所有的Keil、IAR工程文件都放在bsp中各个子目录下，如果是命令行编译也需要在这些目录下进行)

目录名	描述
bsp	针对各个具体开发板、平台的目录，其中包含相应的Keil工程文件（如果包含了Keil MDK的移植）
filesystem	包含了RT-Thread的文件系统组件代码
finsh	包含了RT-Thread的finsh shell组件代码
include	包含了RT-Thread核心的头文件
libc	面向GCC移植的小型C库
libcpu	面向各个芯片移植的代码
net	包含了RT-Thread的网络组件代码
rtgui	包含了RT-Thread/GUI图形用户界面组件代码
src	包含了RT-Thread内核核心的代码

在目录\bsp\sam7s下，有一个project.Uv2文件



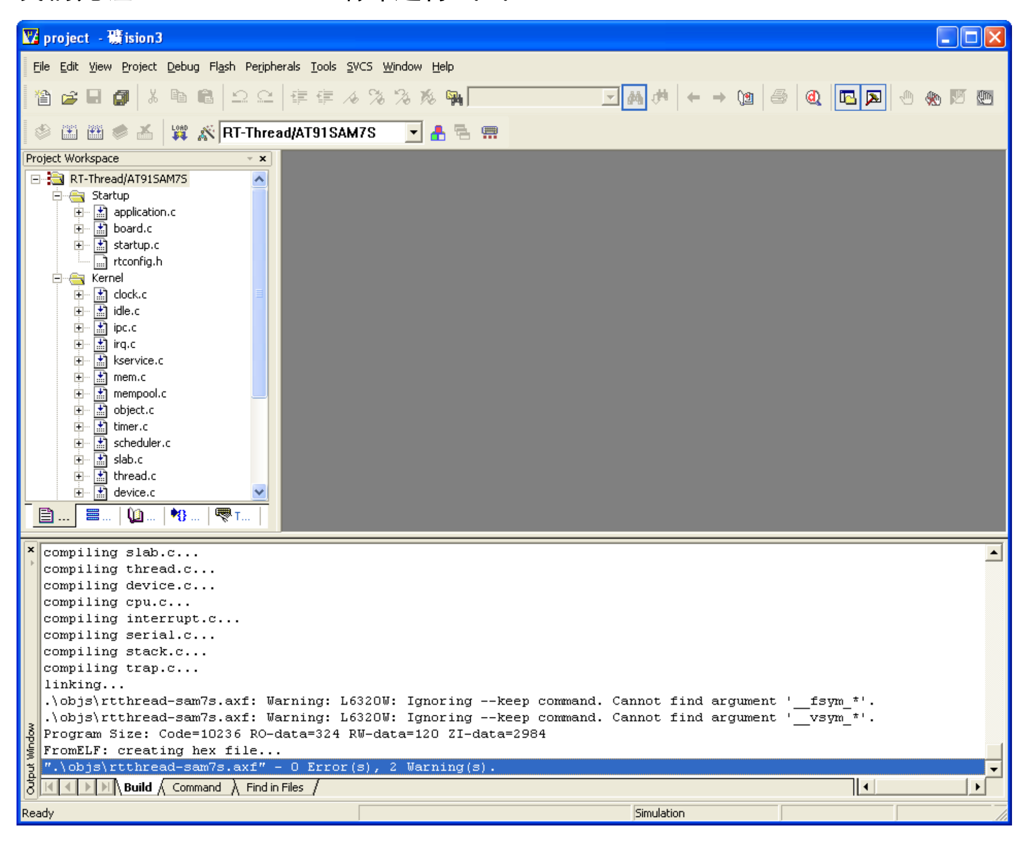
它是一个RealView MDK的工程文件，如果按照上节的步骤正确的安装了RealView MDK，那么在这里直接双击鼠标可以打开这个文件。打开后会出现如下的画面：



这个就是RT-Thread工程文件夹画面，在工程文件列表中总共存在如下几个组

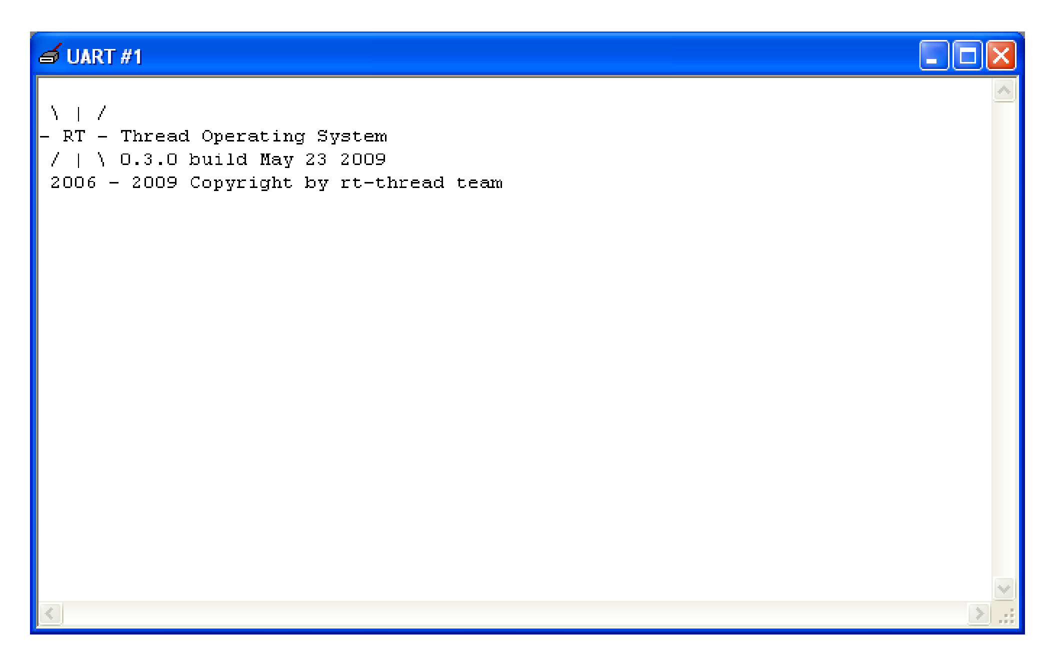
- Startup 用户开发板相关文件及启动文件（对应kernelbsp目录）
- Kernel RT-Thread内核核心实现（对应kernelsrc目录）
- AT91SAM7S 针对ATMEL AT91SAM7S64移植的代码（对应kernellibcpuAT91SAM7S目录）

我们先让RealView MDK编译运行试试：



没什么意外，最后会出现类似画面上的结果，虽然有一些警告但关系不大。

在编译完RT-Thread/AT91SAM7S后，我们可以通过RealView MDK的模拟器来仿真运行RT-Thread。模拟运行的结果如下图所示。



因为用户代码是空的，所以只是显示了RT-Thread的LOGO。

3.3 系统启动代码

一般了解一份代码大多从启动部分开始，同样这里也采用这种方式，先寻找启动的源头：因为RealView MDK的用户程序入口采用了main()函数，所以先看看main()函数在哪个文件中：startup.c，它位于Startup组中，是在AT91SAM7S64的启动汇编代码（启动汇编在AT91SAM7S Group的start_rvds.S中，在后面章节的移植一节中会详细讨论）后跳转到C代码的入口位置。

```
int main (void)
{
    /* 调用RT-Thread的启动函数，rtthread_startup */
    rtthread_startup();

    return 0;
}
```

很简单，main()函数仅仅调用了rtthread_startup()函数。RT-Thread因为支持多种平台，多种编译器，rtthread_startup()函数是RT-Thread的统一入口点。从rtthread_startup()函数中我们将可以看到RT-Thread的启动流程：

```
/* 这个函数将启动RT-Thread RTOS */
void rtthread_startup(void)
{
    /* 初始化硬件中断控制器 */
    rt_hw_interrupt_init();

    /* 初始化硬件开发板 */
    rt_hw_board_init();

    /* 显示RT-Thread的LOGO及版本号 */
```

```
    rt_show_version();

    /* 初始化系统节拍，用于操作系统的时间技术 */
    rt_system_tick_init();

    /* 初始化系统对象 */
    rt_system_object_init();

    /* 初始系统定时器 */
    rt_system_timer_init();

    /*
     * 如果定义了宏RT_USING_HEAP，即RT-Thread使用动态堆
     * AT91SAM7S64的SRAM总共是16kB，地址范围是
     *   0x200000 - 0x204000
     * 所以在调用rt_system_heap_init函数时的最后一个参数是尾地址0x204000
     * 前面的初始地址则根据编译器环境的不同而略微不同
     */
#ifdef RT_USING_HEAP
#ifdef __CC_ARM
    rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)0x204000);
#elif __ICCARM__
    rt_system_heap_init(__segment_end("HEAP"), (void*)0x204000);
#else
    rt_system_heap_init((void*)&__bss_end, (void*)0x204000);
#endif
#endif

    /* 初始化系统调度器 */
    rt_system_scheduler_init();

    /* 如果使用了钩子函数，把rt_hw_led_flash函数挂到idle线程的执行中去 */
#ifdef RT_USING_HOOK
    /* set idle thread hook */
    rt_thread_idle_sethook(rt_hw_led_flash);
#endif

    /* 如果使用了设备框架 */
#ifdef RT_USING_DEVICE
    /* 注册/初始化硬件串口 */
    rt_hw_serial_init();
    /* 初始化所有注册了的设备 */
    rt_device_init_all();
#endif

    /* 初始化上层应用 */
    rt_application_init();

    /* 如果系统中使用了shell系统 */
#ifdef RT_USING_FINSH
    /* 初始化finsh */
    finsh_system_init();
    /* finsh的输入设备是uart1设备 */
```

```
    finsh_set_device("uart1");
#endif

    /* 初始化idle线程 */
    rt_thread_idle_init();

    /* 启动调度器，将进行系统的第一次调度 */
    rt_system_scheduler_start();

    /* 这个地方应该是永远都不应该达到的 */
    return ;
}
```

这部分启动代码，可以分为几个部分：

- 初始化系统相关的硬件
- 初始化系统组件，例如定时器，调度器
- 初始化系统设备，这个主要是为RT-Thread的设备框架做的初始化
- 初始化各个应用线程，并启动调度器

3.4 用户入口代码

上面的启动代码基本上可以说都是和RT-Thread系统相关的，那么用户如何加入自己应用程序的初始化代码：

```
/* 初始化上层应用 */
rt_application_init();
```

这里，用户代码入口位置是rt_application_init()，在这个函数中可以初始化用户应用程序的线程，当后面打开调度器后，用户线程也将得到执行。

在工程中，rt_application_init()的实现在application.c文件中，目前跑的RT-Thread是最简单的，仅包含一个空的rt_application_init()实现：

```
/* 包含RT-Thread的头文件，每一个需要用到RT-Thread服务的文件都需要包含这个文件 */
#include <rtthread.h>

/* 用户应用程序入口点 */
int rt_application_init()
{
    return 0;
}
```

空的实现就意味着，系统中不存在用户的代码，系统只会运行和系统相关的一些代码。在以后的例子中，如果没有特殊的说明，我们都将在这个文件中实现代码，并在rt_application_init()函数中进行初始化工作。

3.5 跑马灯的例子

对于从事电子方面开发的技术工程师来说，跑马灯大概是最简单的例子，就类似于每种编程语言中的Hello World。所以第一个例子就从跑马灯例子开始：创建一个线程，让它不定时的对LED进行更新（关或灭）

```
/* 因为要使用RT-Thread的线程服务，需要包含RT-Thread的头文件 */
#include <rtthread.h>

/* 线程用到的栈，由于ARM是4字节对齐的，所以栈的空间必须是4字节对齐 */
static char led_thread_stack[512];

/* 线程的TCB控制块 */
static struct rt_thread led_thread;

/* 线程的入口点，当线程运行起来后，它将从这里开始执行 */
static void led_thread_entry(void* parameter)
{
    int i;

    /* 这个线程是一个永远循环执行的线程 */
    while (1)
    {
        /* 开LED，然后延时10个OS Tick */
        led_on();
        rt_thread_delay(10);

        /* 关LED，然后延时10个OS Tick */
        led_off();
        rt_thread_delay(10);
    }
}

/* 用户应用程序入口点 */
int rt_application_init()
{
    /*
     * 初始化一个线程
     * 名称是`led`
     * 入口位置是led_thread_entry
     * 入口参数是RT_NULL，这个参数会传递给入口函数的，可以是一个指针或一个数
     * 优先级是25（AT91SAM7S64配置的最大优先级数是32，这里使用25）
     * 时间片是8（如果有相同优先级的线程存在，时间片才会真正起作用）
     */
    rt_thread_init(&led_thread,
        "led",
        led_thread_entry, RT_NULL,
        &led_thread_stack[0], sizeof(led_thread_stack),
        25, 8);

    /*
     * 上一步仅仅是初始化一个线程，也就是为一个线程的运行做准备，
```

```
     * 这里则是启动这个线程
     *
     * 注：这个函数并不代表线程立刻就运行起来，当调度器启动起来后，
     * 线程才得到真正的调度。如果此时，调度器已经运行了，那么则取决于新启
     * 动的线程优先级是否高于当前任务优先级，如果高于，则立刻执行新线程。
     */
    rt_thread_startup(&led_thread);
    return 0;
}
```

在代码中rt_thread_delay(10)函数的作用是延时一段时间，即让led线程休眠10个tick（按照rtconfig.h中的配置，1秒 = RT_TICK_PER_SECOND个tick = 100 tick，即这份代码中是延时100ms）。在休眠的这段时间内，如果没有其他线程运行，操作系统会切换到idle线程运行。

3.6 生产者消费者问题

生产者消费者问题是操作系统中的一个经典问题，在嵌入式操作系统中也经常能够遇到，例如串口中接收到数据，然后由一个任务统一的进行数据的处理：串口产生数据，任务作为一个消费者消费数据。

在下面的例子中，将用RT-Thread的编程模式来实现一个解决生产者、消费者问题的代码。

```
#include <rtthread.h>

/* 定义最大5个元素能够被产生 */
#define MAXSEM 5

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;
/* 指向生产者、消费者在array数组中的读写位置 */
static rt_uint32_t set, get;

/* 生成者线程入口 */
void producer_thread_entry(void* parameter)
{
    int cnt = 0;

    /* 运行100次 */
    while( cnt < 100)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改array内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set%MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n", array[set%MAXSEM]);
        set++;
```

```
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_delay(50);
    }

    rt_kprintf("the producer exit!\n");
}

/* 消费者线程入口 */
void consumer_thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t sum;

    /* 第n个线程，由入口参数传进来 */
    no = (rt_uint32_t)parameter;

    while(1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get%MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", no, array[get%MAXSEM] );
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到100个数目，停止，消费者线程相应停止 */
        if (get == 100) break;

        /* 暂停一小会时间 */
        rt_thread_delay(10);
    }

    rt_kprintf("the consumer[%d] sum is %d \n ", no, sum);
    rt_kprintf("the consumer[%d] exit!\n");
}

/**
 * 用户入口代码
 */
int rt_application_init()
{
```

```
    rt_thread_t p, s;

    /* 初始3个信号量 */
    rt_sem_init(&sem_lock , "lock",     1,      RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty",    MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full , "full",     0,      RT_IPC_FLAG_FIFO);

    /* 创建 生产者 线程 */
    p = rt_thread_create("p",
        producer_thread_entry, RT_NULL,
        1024, 18, 5);
    rt_thread_startup(p);

    /* 创建 消费者 线程，入口相同，入口参数不同，优先级相同 */
    s = rt_thread_create("s1",
        consumer_thread_entry, (void *)1,
        1024, 20, 5);
    if (s != RT_NULL) rt_thread_startup(s);

    s = rt_thread_create("s2",
        consumer_thread_entry, (void *)2,
        1024, 20, 5);
    if (s != RT_NULL) rt_thread_startup(s);

    return 0;
}
```

在上面的例子中，系统：

- 启动了一个生产者线程p，用于向仓库（array数组）中产生一个整数（1 到 100）；
- 启动了两个消费者线程s1和s2，它们的入口函数是同一个，只是通过入口参数分辨它们是第一个消费者线程还是第二个消费者线程。

这两个消费者线程将同时从仓库中获取生成的整数，然后把它打印出来（消费）。

CHAPTER

FOUR

RT-THREAD简介

RT-Thread是一个开放源代码的实时操作系统，并且商业许可证非常宽松的实时操作系统。下图是RT-Thread及外围组件的基本框架图：

FinSH Shell

RT-Device Filesystem
RT-设备虚拟文件系统

LwIP
轻型TCP/IP协议栈

RT-Thread/GUI
图形用户界面

Kernel Library
内核: kservice.c

RT-Thread Kernel 实时内核: src
Object Management 对象管理: object.c
Real time Scheduler 实时调度器: schedule.c
Thread Management 线程管理: thread.c
Internal Thread Communication 线程间通信: ipc.c
Clock Management 时钟管理: clock.c timer.c
Device Driver IO 设备驱动: device.c

CPU Architecture 芯片移植: libcpu
Board Support Package 板级支持包: bsp

Hardware 硬件, CPU/SRAM/Flash/UART/EMAC etc

RT-Thread Kernel内核部分包括了RT-Thread的核心代码，包括对象管理器，线程管理及调度，线程间通信等的微小内核实现（最小能够到达4k ROM，1k RAM体积占用）。内核库是为了保证内核能够独立运作的一套小型类似C库实现（这部分根据编译器自带C库的情况会有些不同，使用GCC编译器时，携带更多的标准C库实现）。CPU及板级支持包包含了RT-Thread支持的各个平台移植代码，通常会包含两个汇编文件，一个是系统启动初始化文件，一个是线程进行上下文切换的文件，其他的都是C源文件。

4.1 实时内核

4.1.1 任务/线程调度

在RT-Thread中线程是最小的调度单位，线程调度算法是基于优先级的全抢占式多线程调度算法，支持256个线程优先级（也能通过配置文件更改为最大支持32个或8个线程优先级），0优先级代表最高优先级，255优先级留给空闲线程使用；支持创建相同优先级线程，相同优先级的线程采用可设置时间片的轮转调度算法；调度器寻找下一个最高优先级就绪线程的时间是恒定的(时间复杂度是1，即O(1))。系统不限制线程数量的多少，只和硬件平台的具体内存相关。

4.1.2 任务同步机制

系统支持信号量、互斥锁作为线程间同步机制。互斥锁采用优先级继存方式以解决优先级翻转问题。信号量的释放动作可安全用于中断服务例程中。同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥锁。

4.1.3 任务间通信机制

系统支持事件、邮箱和消息队列等通信机制。事件支持多事件”或触发”及”与触发”，适合于线程等待多个事件情况。邮箱中一封邮件的长度固定为4字节，效率较消息队列更为高效。通信设施中的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取。

4.1.4 时间管理

系统使用时钟节拍来完成同优先级任务的时间片轮转调度；线程对内核对象的时间敏感性是通过系统定时器来实现的；定时器支持软定时器及硬定时器（软定时器的处理在系统线程的上下文中，硬定时器的处理在中断的上下文中）；定时器支持一次性超时及周期性超时。

4.1.5 内存管理

系统支持静态内存池管理及动态内存堆管理。从静态内存池中获取内存块时间恒定，当内存池为空时，可把申请内存块的线程阻塞(或立刻返回，或等待一段时间后仍未获得内存块返回。这取决于内存块申请时设置的等待时间)，当其他线程释内存块到内存池时，将把相应阻塞线程唤醒。动态堆内存管理对于不同的系统资源情况，提供了面向小内存系统的管理算法及大内存系统的SLAB内存管理算法。

4.1.6 设备管理

系统实现了按名称访问的设备管理子系统，可按照统一的API界面访问硬件设备。在设备驱动接口上，根据嵌入式系统的特点，对不同的设备可以挂接相应的事件，当设备事件触发时，通知给上层的应用程序。

4.2 虚拟文件系统

RT-Thread提供的文件系统称为设备文件系统，它主要包含了一个非常轻型的虚拟文件系统。虚拟文件系统的好处就是，不管下层采用了什么文件系统，例如内存虚拟文件系统，FAT32文件系统还是YAFFS2闪存文件系统，对上层应用程序提供的接口都是统一的。

4.3 轻型IP协议栈

LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的Adam Dunkels等开发的一套用于嵌入式系统的开放源代码TCP/IP协议栈，它在包含完整的TCP协议实现基础上实现了小型的资源占用，因此它十分适合于使用到嵌入式设备中，RT-Thread 采用 LwIP 做为默认的TCP/IP 协议栈，同时根据小型设备的特点对其进行再优化，体积相对进一步减小， **RAM** 占用缩小到**5kB**附近 （依据上层应用使用情况会有浮动）。

4.4 shell系统

RT-Thread的shell系统——FinSH，提供了一套供用户在命令行操作的接口，主要用于调试、查看系统信息。由于系统程序大多数采用C语言来编写，FinSH命令行的设计被设计成类似C语言表达式的风格：它能够解析执行大部分C语言的表达式，也能够使用类似于C语言的函数调用方式（或函数指针方式）访问系统中的函数及全局变量。

4.5 图形用户界面

RT-Thread/GUI组件是一套完全针对嵌入式系统而进行优化的图形用户界面，它在保留通常意义的多窗口的前提下，提出了面板，工作台，视图的概念，通过一个个视图的渲染展现出图形用户界面绚丽的外观。它同样也包括了基本控件的支持、中文显示的支持、多线程的支持；针对嵌入式系统计算能力不足的特点，它会自动对界面区域进行可视区域的剪切，该重绘显示的地方进行重绘，被覆盖的地方则不进行重复绘图。

4.6 支持的平台

芯片	核心	描述	编译环境
AT91SAM7X	ARM7TDMI	基本系统，TCP/IP协议栈	GNU GCC及RealView MDK
LPC2478	ARM7TDMI	基本系统，文件系统，TCP/IP协议栈	GNU GCC及RealView MDK
s3c2440	ARM920T	基本系统，TCP/IP协议栈，文件系统，图形界面，面向0.4.x分支	GNU GCC及RealView MDK
STM32	ARM Cortex M3	基本系统，TCP/IP协议栈，文件系统，图形界面(仅STM32F103ZE支持)	GNU GCC及RealView MDK、IAR ARM
LM3S	ARM Cortex M3	基本系统，TCP/IP协议栈，文件系统	GNU GCC及RealView MDK
i386	x86	基本系统（运行于真实机器或QEMU）	Linux，GNU GCC

CHAPTER

FIVE

内核对象模型

RT-Thread的内核对象模型是一种非常有趣的面向对象实现方式。由于C语言更为面向系统底层，操作系统核心通常都是采用C语言和汇编语言混合编写而成。C语言作为一门高级计算机编程语言，一般被认为是一种面向过程的编程语言：程序员按照特定的方式把要处理事物的过程一级级分解成一个个子过程。

面向对象源于人类对世界的认知多偏向于类别模式，根据世界中不同物品的特性分门别类的组织在一起抽象并归纳，形成各个类别的自有属性。在计算机领域一般采用一门新的，具备面向对象特征的编程语言实现面向对象的设计，例如常见的编程语言C++，Java，Python等。那么RT-Thread既然有意引入对象系统，为什么不直接采用C++来实现？这个需要从C++的实现说起，用过C++的开发人员都知道，C++的对象系统中会引入很多未知的东西，例如虚拟重载表，命名粉碎，模板展开等。对于一个需要精确控制的系统，这不是一个很好的方式，假于它人之手不如握入己手！

面向对象有它非常优越的地方，取其精华(即面向对象思想，面向对象设计)，也就是RT-Thread内核对象模型的来源。RT-Thread实时操作系统中包含一个小型的，非常紧凑的对象系统，这个对象系统完全采用C语言实现。在了解RT-Thread内部或采用RT-Thread编程时有必要先熟悉它，它是RT-Thread实现的基础。

5.1 C语言的对象化模型

面向对象的特征主要包括：

- 封装，隐藏内部实现
- 继承，复用现有代码
- 多态，改写对象行为

采用C语言实现的关键是如何运用C语言本身的特性来实现上述面向对象的特征。

5.1.1 封装

封装是一种信息隐蔽技术，它体现于类的说明，是对象的重要特性。封装使数据和加工该数据的方法（函数）封装为一个整体，以实现独立性很强的模块，使得用户只能见到对象的外特性（对象能接受哪些消息，具有那些处理能力），而对象的内特性（保存内部状态的私有数据和实现加工能力的算法）对用户是隐蔽的。封装的目的在于把对象的设计者和对象者的使用分开，使用者不必知晓行为实现的细节，只须用设计者提供的消息来访问该对象。

在C语言中，大多数函数的命名方式是动词+名词的形式，例如要获取一个semaphore，会命名成take_semaphore，重点在take这个动作上。在RT-Thread系统的面向对象编程中刚好相反，命名为rt_sem_take，即名词+动词的形式，重点在名词上，体现了一个对象的方法。另外对于某些方法，仅局限在对象内部使用，它们将采用static修辞把作用范围局限在一个文件的内部。通过这样的方式，把一些不想让用户知道的信息屏蔽在封装里，用户只看到了外层的接口，从而形成了面向对象中的最基本的对象封装实现。

一般属于某个类的对象会有一个统一的创建，析构过程。在RT-Thread中这些分为两类(以semaphore对象为例)：

- 对象内存数据块已经存在，需要对它进行初始化 – rt_sem_init;
- 对象内存数据块还未分配，需要创建并初始化 – rt_sem_create。

可以这么认为，对象的创建(create)是以对象的初始化(init)为基础的，创建动作相比较而言多了个内存分配的动作。

相对应的两类析构方式：

- 由rt_sem_init初始化的semaphore对象 – rt_sem_detach;
- 由rt_sem_create创建的semaphore对象 – rt_sem_delete.

5.1.2 继承

继承性是子类自动共享父类之间数据和方法的机制。它由类的派生功能体现。一个类直接继承其它类的全部描述，同时可修改和扩充。继承具有传递性。继承分为单继承（一个子类只有一父类）和多重继承（一个类有多个父类，当前RT-Thread的对象系统不能支持）。类的对象是各自封闭的，如果没继承性机制，则类对象中数据、方法就会出现大量重复。继承不仅支持系统的可重用性，而且还促进系统的可扩充性。

类似的实现代码如下程序清单：

```
/* 父类 */
struct parent_class
{
    int a, b;
    char *str;
};

/* 继承于父类的子类 */
struct child_class
{
    struct parent_class p;
    int a, b;
};

/* 操作示例函数*/
void func()
{
    struct child_class obj, *obj_ptr; /* 子类对象及指针 */
    struct parent_class *parent_ptr; /* 父类指针 */

    obj_ptr = &obj;
    /* 取父指针 */
```

```
    parent_ptr = (struct parent*) &obj;

    /* 可通过转换过类型的父类指针访问相应的属性 */
    parent_ptr->a = 1;
    parent_ptr->b = 5;

    /* 子类属性的操作 */
    obj_ptr->a = 10;
    obj_ptr->b = 100;
}
```

在上面代码中，注意child_class结构中第一个成员p，这种声明方式代表child_class类型的数据中开始的位置包含一个parent_class类型的变量。在函数func中obj是一个child_class对象，正像这个结构类型指示的，它前面的数据应该包含一个parent_class类型的数据。在第21行的强制类型赋值中parent_ptr指向了obj变量的首地址，也就是obj变量中的p对象。好了，现在parent_ptr指向的是一个真真实实的parent类型的结构，那么可以按照parent的方式访问其中的成员，当然也包括可以使用和parent结构相关的函数来处理内部数据，因为一个正常的，正确的代码，它是不会越界访问parent结构体以外的数据。

经过这基本的结构体层层相套包含，对象简单的继存关系就体现出来了：父对象放于数据块的最前方，代码中可以通过强制类型转换获得父对象指针。

5.1.3 多态

对象根据所接收的消息而做出动作。同一消息为不同的对象接受时可产生完全不同的行动，这种现象称为多态性。利用多态性用户可发送一个通用的信息，而将所有的实现细节都留给接受消息的对象自行决定，如是，同一消息即可调用不同的方法。例如：RT-Thread系统中的设备：抽象设备具备接口统一的读写接口。串口是设备的一种，也应支持设备的读写。但串口的读写操作是串口所特有的，不应和其他设备操作完全相同，例如操作串口的操作不应应用于SD卡设备中。

多态性的实现受到继承性的支持，利用类继承的层次关系，把具有通用功能的协议存放在类层次中尽可能高的地方，而将实现这一功能的不同方法置于较低层次，这样，在这些低层次上生成的对象就能给通用消息以不同的响应。

RT-Thread对象模型采用结构封装中使用指针的形式达到面向对象中多态的效果，例如：

```
/* 抽象父类 */
struct parent_class
{
    int a;

    /* 反映不同类别属性的方法 */
    void (*vfunc)(int a);
}

/* 抽象类的方法调用 */
void parent_class_vfunc(struct parent_class *self, int a)
{
    assert(self != NULL);
    assert(slef->vfunc != NULL);

    /* 调用对象本身的虚拟函数 */
```

```
    self->vfunc(a);
}

/* 继承自parent_class的子类 */
struct child_class
{
    struct parent_class parent;
    int b;
};

/* 子类的构造函数 */
void child_class_init(struct child_class* self)
{
    struct parent_class* parent;

    /* 强制类型转换获得父类指针 */
    parent = (struct base_class*) self;

    assert(parent != NULL);

    /* 设置子类的虚拟函数 */
    parent->vfunc = child_class_vfunc;
}

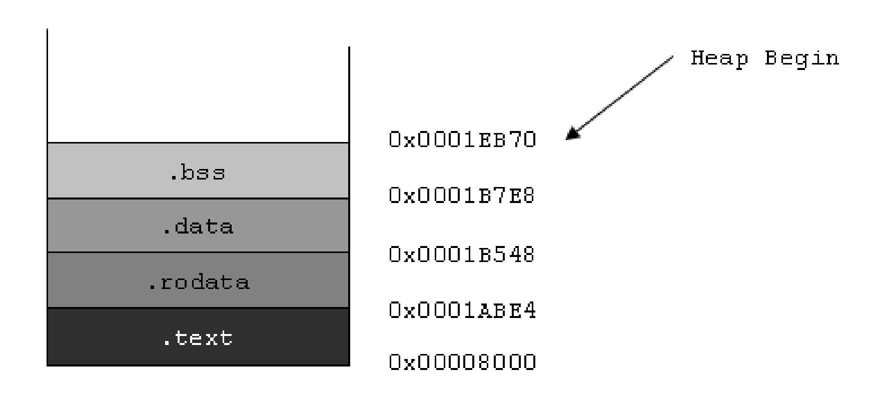
/* 子类的虚拟函数实现 */
static void _child_class_vfunc(struct child_class*self, int a)
{
    self->b = a + 10;
}
```

5.2 内核对象模型

5.2.1 静态对象和动态对象

RT-Thread的内核映像文件在编译时会形成如下图所示的结构（以AT91SAM7S64为例）：其中主要包括了这么几段：

Segment	Description
.text	代码正文段
.data	数据段，用于放置带初始值的全局变量
.rodata	只读数据段，用于放置只读的全局变量（常量）
.bss	bss段，用于放置未初始化的全局变量



上图为AT91SAM7S64运行时内存映像图。当系统运行时，这些段也会相应的映射到内存中。在RT-Thread系统初始化时，通常bss段会清零，而堆（Heap）则是除了以上这些段以外可用的内存空间（具体的地址空间在系统启动时由参数指定），系统运行时动态分配的内存块就在堆的空间中分配出来的，如下代码：

```
rt_uint8_t* msg_ptr;
msg_ptr = (rt_uint8_t*) rt_malloc (128);
rt_memset(msg_ptr, 0, 128);
```

msg_ptr 指向的128字节内存空间就是位于堆空间中的。

而一些全局变量则是存放于.data和.bss段中，.data存放的是具有初始值的全局变量（.rodata可以看成一个特殊的data段，是只读的），如下代码：

```
#include <rtthread.h>

const static rt_uint32_t sensor_enable = 0x000000FE;
rt_uint32_t sensor_value;
rt_bool_t sensor_inited = RT_FALSE;

void sensor_init()
{
    [...]
}
```

sensor_value存放在.bss段中，系统启动后会自动初始化成零。sensor_inited变量则存放在.data段中，而sensor_enable存放在.rodata端中。

在RT-Thread内核对象中分为两类：静态内核对象和动态内核对象。静态内核对象通常放在.data或.bss段中，在系统启动后在程序中初始化；动态内核对象则是从堆中创建的，而后手工做初始化。

RT-Thread中操作系统级的设施都是一种内核对象，例如线程，信号量，互斥量，定时器等。以下的代码所示的即为静态线程和动态线程的例子：

```
/* 线程1的对象和运行时用到的栈 */
static struct rt_thread thread1;
static rt_uint8_t thread1_stack[512];

/* 线程1入口 */
```

```
void thread1_entry(void* parameter)
{
    int i;

    while (1)
    {
        for (i = 0; i < 10; i ++)
        {
            rt_kprintf("%d\n", i);

            /* 延时100个OS Tick */
            rt_thread_delay(100);
        }
    }
}

/* 线程2入口 */
void thread2_entry(void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf("Thread2 count:%d\n", ++count);

        /* 延时50个OS Tick */
        rt_thread_delay(50);
    }
}

/* 用户应用程序入口 */
int rt_application_init()
{
    rt_thread_t thread2_ptr;
    rt_err_t result;

    /* 初始化线程1 */
    result = rt_thread_init(&thread1,
        "thread1",
        thread1_entry, RT_NULL, /* 线程的入口是thread1_entry，参数是RT_NULL */
        &thread1_stack[0], sizeof(thread1_stack), /* 线程栈是thread1_stack */
        200, 10);   /* 优先级是200，时间片是10个OS Tick */
    if (result == RT_EOK) rt_thread_startup(&thread1); /* 启动线程 */

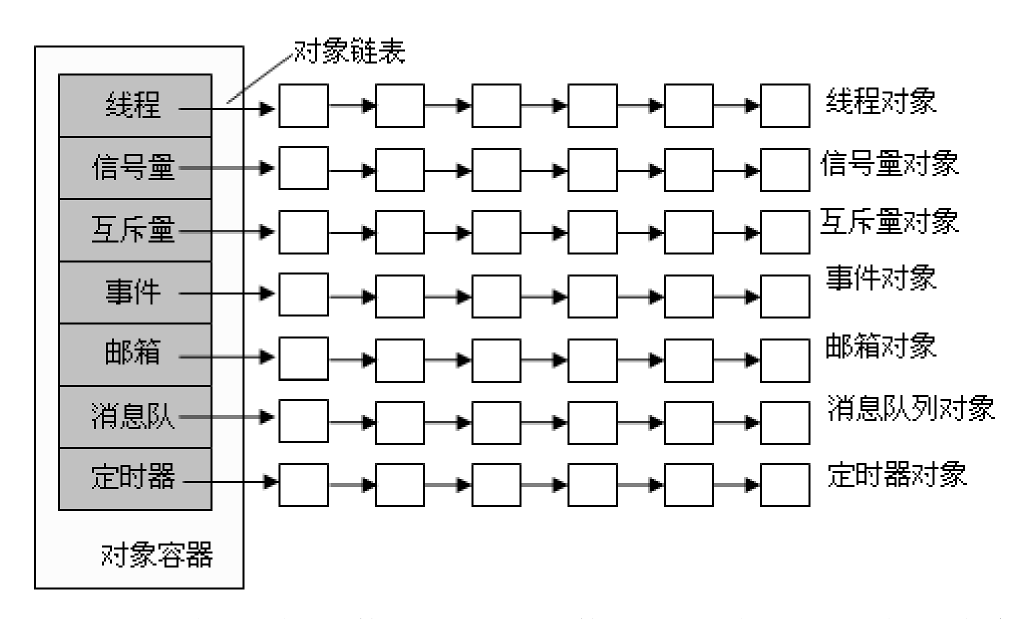
    /* 创建线程2 */
    thread2_ptr = rt_thread_create("thread2",
        thread2_entry, RT_NULL, /* 线程的入口是thread2_entry，参数是RT_NULL */
        512, 250, 25); /* 栈空间是512，优先级是250，时间片是25个OS Tick */
    if (thread2_ptr != RT_NULL) rt_thread_startup(thread2_ptr); /* 启动线程 */

    return 0;
}
```

例子中，thread1是一个静态线程对象，而thread2是一个动态线程对象。thread1对象的内存空间，

包括线程控制块 thread1，栈空间thread1_stack都是编译时决定的，因为代码中都不存在初始值，都统一放在.bss段中。thread2运行中用到的空间都是动态分配的，包括线程控制块（thread2_ptr指向的内容）和栈空间。

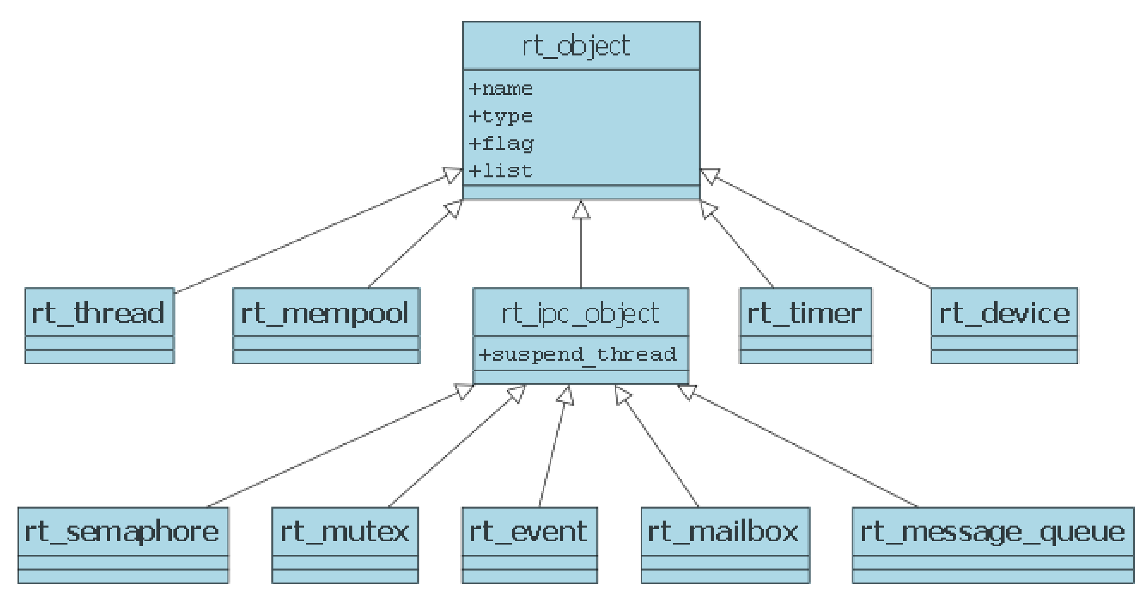
5.2.2 内核对象管理工作模式



RT-Thread采用内核对象管理系统来访问/管理所有内核对象。内核对象包含了内核中绝大部分设施，而这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。通过内核对象系统，RT-Thread做到了不依赖于具体的内存分配方式，系统的灵活性得到极大的提高。

RT-Thread内核对象包括：线程，信号量，互斥锁，事件，邮箱，消息队列和定时器，内存池，设备驱动等。对象容器中包含了每类内核对象的信息，包括对象类型，大小等。对象容器给每类内核对象分配了一个链表，所有的内核对象都被链接到该链表上。

下图显示了RT-Thread中各类内核对象的派生和继承关系。对于每一种具体内核对象和对象控制块，除了基本结构外，还有自己的扩展属性（私有属性），例如，对于线程控制块，在基类对象基础上进行扩展，增加了线程状态、优先级等属性。这些属性在基类对象的操作中不会用到，只有在与具体线程相关的操作中才会使用。因此从面向对象的观点，可以认为每一种具体对象是抽象对象的派生，继承了基本对象的属性并在此基础上扩展了与自己相关的属性。



在对象管理模块中，定义了通用的数据结构，用来保存各种对象的共同属性，各种具体对象只需要在此基础上加上自己的某些特别的属性，就可以清楚的表示自己的特征。这种设计方法的优点：

1. 提高了系统的可重用性和扩展性，增加新的对象类别很容易，只需要继承通用对象的属性再加少量扩展即可。
2. 提供统一的对象操作方式，简化了各种具体对象的操作，提高了系统的可靠性。

5.2.3 对象控制块

```
struct rt_object
{
    /* 内核对象名称     */
    char       name[RT_NAME_MAX];
    /* 内核对象类型     */
    rt_uint8_t  type;
    /* 内核对象的参数   */
    rt_uint8_t  flag;
    /* 内核对象管理链表 */
    rt_list_t   list;
};
```

目前内核对象支持的类型如下：

```
enum rt_object_class_type
{
    RT_Object_Class_Thread = 0,      /* 对象为线程类型        */
#ifdef RT_USING_SEMAPHORE
    RT_Object_Class_Semaphore,       /* 对象为信号量类型      */
#endif
#ifdef RT_USING_MUTEX
    RT_Object_Class_Mutex,           /* 对象为互斥锁类型      */
#endif
#ifdef RT_USING_EVENT
```

```
    RT_Object_Class_Event,          /* 对象为事件类型        */
#endif
#ifdef RT_USING_MAILBOX
    RT_Object_Class_MailBox,        /* 对象为邮箱类型        */
#endif
#ifdef RT_USING_MESSAGEQUEUE
    RT_Object_Class_MessageQueue,   /* 对象为消息队列类型    */
#endif
#ifdef RT_USING_MEMPOOL
    RT_Object_Class_MemPool,        /* 对象为内存池类型      */
#endif
#ifdef RT_USING_DEVICE
    RT_Object_Class_Device,         /* 对象为设备类型        */
#endif
    RT_Object_Class_Timer,          /* 对象为定时器类型      */
    RT_Object_Class_Unknown,        /* 对象类型未知          */
    RT_Object_Class_Static = 0x80   /* 对象为静态对象        */
};
```

5.2.4 内核对象接口

初始化系统对象

在初始化各种内核对象之前，首先需对对象管理系统进行初始化。在系统中，每类内核对象都有一个静态对象容器，一个静态对象容器放置一类内核对象，初始化对象管理系统的任务就是初始化这些对象容器，使之能够容纳各种内核对象，初始化系统对象使用以下接口：

```
void rt_system_object_init(void)
```

以下是对象容器的数据结构：

```
struct rt_object_information
{
    enum rt_object_class_type type; /* 对象类型 */
    rt_list_t object_list;          /* 对象链表 */
    rt_size_t object_size;          /* 对象大小 */
};
```

一种类型的对象容器维护了一个对象链表object_list，所有对于内核对象的分配，释放操作均在该链表上进行。

初始化对象

使用对象前须先对其进行初始化。初始化对象使用以下接口：

```
void rt_object_init(struct rt_object* object, enum rt_object_class_type type, const char* name)
```

对象初始化，实现上就是把对象放入到其相应的对象容器中，即将对象插入到对象容器链表中。

脱离对象

从内核对象管理器中脱离一个对象。脱离对象使用以下接口：

```
void rt_object_detach(rt_object_t object)
```

使用该接口后，静态内核对象将从内核对象管理器中脱离，对象占用的内存不会被释放。

分配对象

在当前内核中，定义了数种内核对象，这些内核对象被广泛的用于线程的管理，线程之间的同步，通信等。因此，系统随时需要新的对象来完成这些操作，分配新对象使用以下接口：

```
rt_object_t rt_object_allocate(enum rt_object_class_type type, const char* name)
```

使用以上接口，首先根据对象类型来获取对象信息，然后从内存堆中分配对象所需内存空间，然后对该对象进行必要的初始化，最后将其插入到它所在的对象容器链表中。

删除对象

不再使用的对象应该立即被删除，以释放有限的系统资源。删除对象使用以下接口：

```
void rt_object_delete(rt_object_t object)
```

使用以上接口时，首先从对象容器中脱离对象，然后释放对象所占用的内存。

查找对象

通过指定的对象类型和对象名查找对象，查找对象使用以下接口：

```
rt_object_t rt_object_find(enum rt_object_class_type type, const char* name)
```

使用以上接口时，在对象类型所对应的对象容器中遍历寻找指定对象，然后返回该对象，如果没有找到这样的对象，则返回空。

辨别对象

判断指定对象是否是系统对象（静态内核对象）。辨别对象使用以下接口：

```
rt_err_t rt_object_is_systemobject(rt_object_t object)
```

通常采用rt_object_init方式挂接到内核对象管理器中的对象是系统对象。

CHAPTER

SIX

线程调度与管理

一个典型的简单程序会设计成一个串行的系统运行：按照准确的指令步骤一次一个指令的运行。但是这种方法对于复杂一些的实时应用是不可行的，因为它们通常需要在固定的时间内“同时”处理多个输入输出，实时软件应用程序应该设计成一个并行的系统。

并行设计需要开发人员把一个应用分解成一个个小的，可调度的，序列化的程序单元。当合理的划分任务，正确的并行执行时，这种设计能够让系统满足实时系统的性能及时间的要求。

6.1 实时系统的需求

如第二章里描述的，实时系统指的是在固定的时间内正确地对外部事件做出响应。这个“时间内”(英文叫做deadline)，系统内部会做一些处理，例如输入数据的分析计算，加工处理等。而在这段时间之外，系统可能会闲下来，做一些空余的事。

例如一个手机终端，当一个电话拨入的时候，系统应当及时发出振铃、声音提示以通知主人有来电，询问是否进行接听。而在非电话拨入的时候，人们可以用它进行一些其它工作，例如听音乐，玩游戏。

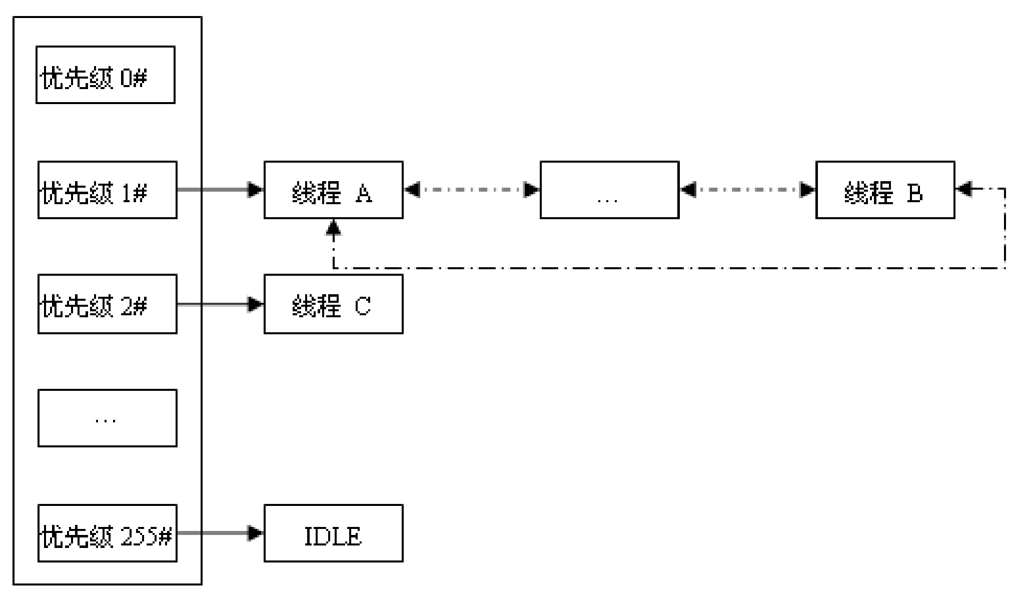
从上面的例子我们可以看出，实时系统是一种需求倾向性的系统，对于实时的事件需要在第一时间内做出回应，而对非实时任务则可以在实时事件到达时为之让路 – 被抢占。所以实时系统也可以看成是一个等级的系统，不同重要性的任务具有不同的优先等级：重要的事件能够优先被相应执行，非重要的事件可以适当往后挪。

在RT-Thread实时操作系统中，任务采用了线程来实现，线程是RT-Thread中的最基本的调度单位，它描述了一个任务执行的上下文关系，也描述了这个任务所处的优先等级。重要的任务能拥有相对较高的优先级，非重要的任务优先级可以放低，并且可以类似Linux一样具备分时的效果。

6.2 线程调度器

RT-Thread中提供的线程调度器是基于全抢占式优先级的调度，在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。系统总共支持256个优先级(0 ～ 255，数值越小的优先级越高，0为最高优先级，255分配给空闲线程使用，一般用户不使用。在一些资源比较紧张的系统中，可以根据情况选择只支持8个或32个优先级的系统配置)。在系统中，当有比当前线程优先级还要高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理机进行执行。

如下图所示，RT-Thread调度器实现中包含一组，总共256个优先级队列数组(如果系统最大支持32个优先级，那么这里将是32个优先级队列数组)，每个优先级队列采用双向环形链表的方式链接，255优先级队列中一般只包含一个idle线程。



在优先级队列1#和2#中，分别有线程A – 线程C。由于线程A、B的优先级比线程C的高，所以此时线程C得不到运行，必须要等待线程A – 线程B（阻塞）都让出处理机后才能得到执行。

当一个操作系统仅仅具备了高优先级任务能够“立马”获得处理机进行执行，它依然不算是实时操作系统：例如线程A、B、C，假设线程C的优先级最低（线程A的优先级比线程B的优先级高），并且处于就绪运行状态。线程A和线程B处于等待某一事件的到达而挂起的状态。当线程B因为事件触发而唤醒变成就绪执行状态时，在线程B运行的一瞬间，线程A等待的事件亦到达，使得线程A变成就绪的状态。那么接下去，按照优先级高先调度的原则，应该线程A得到执行。

当进行下一个就绪线程抉择时，抉择时间的长短将极大的影响到系统的实时性：所有就绪线程都链接在优先级数组中，抉择过程演变为在优先级数组中寻找最高优先级线程。比较一般的方法是，从优先级数组的0元素位置开始，依次寻找。当就绪线程处于末位时，比较次数无疑大幅上升，所以这也不是一个时间可预测的算法。RT-Thread实时中采用了基于位图的优先级算法（时间复杂度O(1)），通过位图的定位（花费的时间与优先级无关），快速的获得优先级最高的线程。

RT-Thread实时操作中也允许创建相同优先级的线程。相同优先级的线程采用时间片轮转进行调度（也就是通常说的分时调度器）。如上图例中所示的线程A 和线程B，假设它们一次最大允许运行的时间片分别是10个时钟节拍和7个时钟节拍。那么线程B的运行需要在线程A运行完它的时间片（10个时钟节拍）后才能获得运行（如果中途线程A被挂起了，线程B因为变成系统中就绪线程中最高的优先级，会马上获得运行）。

因为RT-Thread调度器的实现是采用优先级链表的方式，所以系统中的总线程数不受限制，只和系统所能提供的内存资源相关。为了保证系统的实时性，系统尽最大可能地保证高优先级的线程得以运行。线程调度的原则是一旦任务状态发生了改变，并且当前运行的线程优先级小于优先级队列组中线程最高优先级时，立刻进行线程切换（除非当前系统处于中断处理程序中或禁止线程切换的状态）。

6.3 线程控制块

线程控制块是操作系统用于控制线程的一个数据结构，它会存放线程的一些信息，例如优先级，线程名称等，也包含线程与线程之间的链表结构，线程等待事件集合等。

在RT-Thread实时操作系统中，线程控制块由结构体struct rt_thread（如下代码中所示）表示，另外一种写法是rt_thread_t，表示的是线程的句柄，在C的实现上是指向线程控制块的指针。

```
typedef struct rt_thread* rt_thread_t;

struct rt_thread
{
    /* rt object */
    char        name[RT_NAME_MAX];      /* 对象的名称 */
    rt_uint8_t  type;                   /* 对象的类型 */
    rt_uint8_t  flags;                  /* 对象的参数 */

    rt_list_t   list;                   /* 对象链表 */

    rt_thread_t tid;                    /* 线程ID */
    rt_list_t   tlist;                  /* 线程链表 */

    /* 栈指针和线程入口 */
    void*       sp;                      /* 线程的栈指针 */
    void*       entry;                   /* 线程的入口位置 */
    void*       parameter;               /* 入口参数 */
    void*       stack_addr;             /* 栈起始地址 */
    rt_uint16_t stack_size;             /* 栈大小 */

    rt_err_t    error;                  /* 线程运行过程的错误号 */

    /* priority */
    rt_uint8_t  current_priority;       /* 线程当前的优先级 */
    rt_uint8_t  init_priority;          /* 线程的初始优先级 */
#if RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t  number;
    rt_uint8_t  high_mask;
#endif
    rt_uint32_t number_mask;

#if defined(RT_USING_EVENT) || defined(RT_USING_FASTEVENT)
    /* 线程事件 */
    rt_uint32_t event_set;
    rt_uint8_t  event_info;
#endif

    rt_uint8_t  stat;                   /* 线程状态 */

    rt_ubase_t  init_tick;              /* 线程初始分配的时间片 */
    rt_ubase_t  remaining_tick;         /* 线程当前运行时剩余的时间片 */

    struct rt_timer thread_timer;       /* 线程定时器 */
```

```
    rt_uint32_t user_data;              /* 用户数据 */
};
```

最后的一个成员user_data可由用户挂接一些数据信息到线程控制块中，以提供类似线程私有数据的实现。

6.4 线程状态

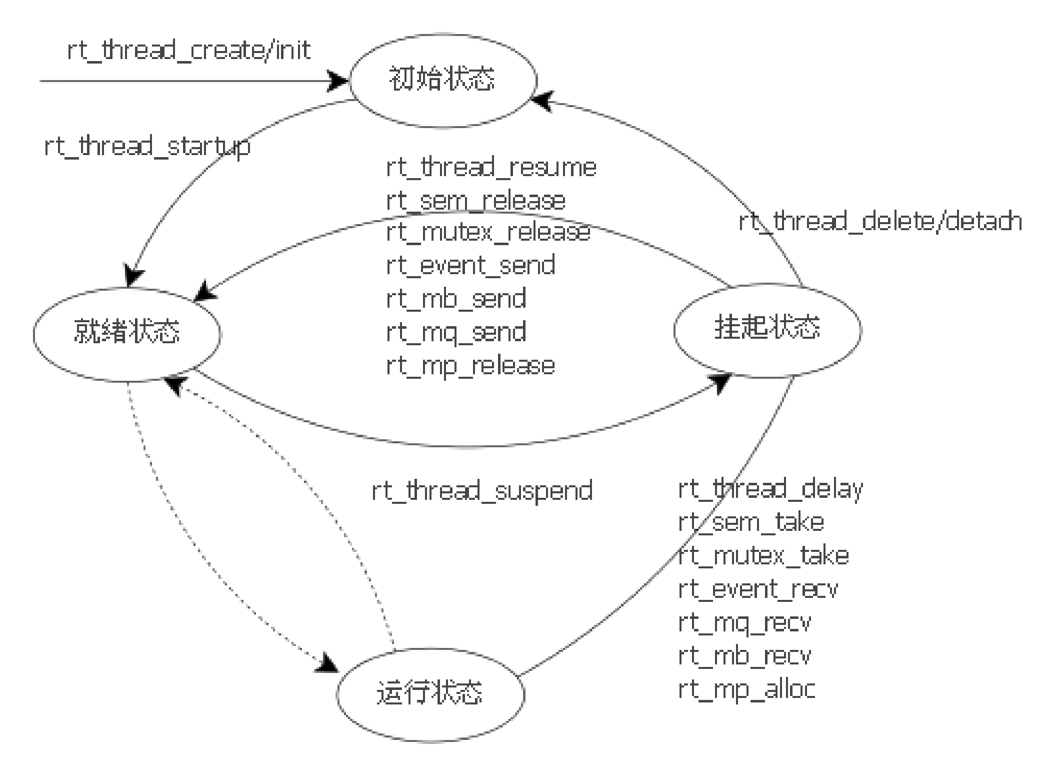
线程运行的过程中，在一个时间内只允许一个线程在处理器中运行，即线程会有多种不同的线程状态，如运行态，非运行态等。在RT-Thread实时操作系统中，线程包含四种状态，操作系统会自动根据它运行的情况而动态调整它的状态。

RT-Thread中的四种线程状态：

状态	描述
RT_THREAD_INIT/CLOSE	线程初始状态。当线程刚开始创建还没开始运行时就处于这个状态；当线程运行结束时也处于这个状态。在这个状态下，线程不参与调度
RT_THREAD_SUSPEND	挂起态。线程此时被挂起：它可能因为资源不可用而等待挂起；或主动延时一段时间而被挂起。在这个状态下，线程不参与调度
RT_THREAD_READY	就绪态。线程正在运行；或当前线程运行完让出处理机后，操作系统寻找最高优先级的就绪态线程运行
RT_THREAD_RUNNING	运行态。线程当前正在运行，在单核系统中，只有rt_thread_self()函数返回的线程处于这个状态；在多核系统中则不受这个限制。

RT-Thread实时操作系统提供一系列的操作系统调用接口，使得线程的状态在这四个状态之间来回的变换。例如一个就绪态的线程由于申请一个资源(例如使用rt_sem_take)，而有可能进入阻塞态。又如，一个外部中断发生，转入中断处理函数，中断处理函数释放了某个资源，导致了当前运行任务的切换，唤醒了另一阻塞态的任务，改变其状态为就绪态等等。

几种状态间的转换关系如下图所示：



线程通过调用函数rt_thread_create/init调用进入到初始状态（RT_THREAD_INIT/RT_THREAD_CLOSE），通过函数rt_thread_startup调用后进入到就绪状态（RT_THREAD_READY）。当这个线程调用rt_thread_delay，rt_sem_take，rt_mb_recv等函数时，将主动挂起或由于获取不到资源进入到挂起状态（RT_THREAD_SUSPEND）。在挂起状态的线程，如果它等待超时依然未获得资源或由于其他线程释放了资源，它将返回到就绪状态。

6.5 空闲线程

空闲线程是系统线程中一个比较特殊的线程，它具备最低的优先级，当系统中无其他线程可运行时，调度器将调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起。在RT-Thread实时操作系统中空闲线程提供了钩子函数，可以让系统在空闲的时候执行一定任务，例如系统运行指示灯闪烁，电源管理等。

6.6 调度器相关接口

6.6.1 调度器初始化

在系统启动时需要执行调度器的初始化，以初始化调度器用到的一些全局变量。调度器初始化可以调用以下接口。

```
void rt_system_scheduler_init(void)
```

6.6.2 启动调度器

在系统完成初始化后切换到第一个线程，可以调用如下接口。

```
void rt_system_scheduler_start(void)
```

在调用这个函数时，它会查找系统中优先级最高的就绪态线程，然后切换过去运行。注：在调用这个函数前，必须先做idle线程的初始化，即保证系统至少能够找到一个就绪状态的线程进行执行。此函数是永远不会返回的。

6.6.3 执行调度

让调度器执行一次线程的调度可通过如下接口。

```
void rt_schedule(void)
```

调用这个函数后，系统会计算一次系统中就绪态的线程，如果存在比当前线程更高优先级的线程时，系统将切换到高优先级的线程去。通常情况下，用户不需要直接调用这个函数。

注：在中断服务例程中也可以调用这个函数，如果满足任务切换的条件，它会记录下中断前的线程及需要切换到的更高优先级线程，在中断服务例程处理完毕后执行真正的线程上下文切换（即中断中的线程上下文切换），最终切换到目标线程去。

6.6.4 设置调度器钩子

整个系统的运行基本上都处于一个线程运行状态、中断触发响应中断、切换到其他线程，甚至是线程间的切换的过程中。有时用户可能会想知道在一个时刻发生了什么样的线程切换，可以通过调用下面的函数接口设置一个相应的钩子函数。在系统线程切换时，这个钩子函数将被调用。

```
void rt_scheduler_sethook(void (*hook)(struct rt_thread* from, struct rt_thread* to))
```

调用这个函数可设置一个声明成如下形式

```
void hook(struct rt_thread* from, struct rt_thread* to);
```

的函数作为系统线程切换时被自动调用的钩子函数。其中参数，from、to分别指出了切换到和切换出线程的控制块指针。

Note: 请仔细编写你的钩子函数，如有不甚将很可能导致整个系统运行不正常。

6.7 线程相关接口

6.7.1 线程创建

一个线程要成为可执行的对象就必须由操作系统内核来为它创建/初始化一个线程句柄。可以通过如下的接口来创建一个线程。

```
rt_thread_t rt_thread_create (const char* name,
        void (*entry)(void* parameter), void* parameter,
        rt_uint32_t stack_size,
        rt_uint8_t priority, rt_uint32_t tick)
```

在调用这个函数时，需要为线程指定名称，线程入口位置，入口参数，线程栈大小，优先级及时间片大小。线程名称的最大长度由宏RT_NAME_MAX指定，多余部分会被自动截掉。栈大小的单位是字节，在大多数系统中需要做对齐（例如ARM体系结构中需要向4字节对齐）。线程的优先级范围根据系统配置情况，如果支持256级线程优先级，那么范围是从0 ～ 255，数值越小优先级越高0。时间片(tick) 的单位是操作系统的时钟节拍，当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度，这段时间片运行结束后，调度器自动选择下一个就绪的同优先级线程进行运行。

Note: 确定一个线程的栈空间大小，是一件令人繁琐的事情，在RT-Thread中，可以先指定一个稍微大的栈空间，例如1024或2048，然后在FinSH shell中查看线程运行的过程中线程使用栈的最大值(在RT-Thread中，会记录一个线程使用过程中使用到的栈的最大深度，通过finsh shell命令list_thread()可以看到从系统启动时，到当前时间点，线程使用的最大栈深度)。

调用这个函数后，系统会从动态堆内存中分配一个线程句柄（即TCB，线程控制块）以及按照参数中指定的栈大小从动态堆内存中分配相应的空间。

创建一个线程的例子如下代码所示。

```
/*
 * 程序清单：动态线程
 *
 * 这个程序会初始化2个动态线程，它们拥有共同的入口函数，但参数不相同
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程入口 */
static void thread_entry(void* parameter)
{
        rt_uint32_t count = 0;
        rt_uint32_t no = (rt_uint32_t) parameter; /* 获得正确的入口参数 */

        while (1)
        {
                /* 打印线程计数值输出 */
                rt_kprintf("thread%d count: %d\n", no, count ++);

                /* 休眠10个OS Tick */
                rt_thread_delay(10);
        }
}

int thread_dynamic_simple_init()
{
        /* 创建线程1 */
        tid1 = rt_thread_create("thread",
                thread_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
                THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
        if (tid1 != RT_NULL)
                rt_thread_startup(tid1);
```

```
36          else
37                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
38
39          /* 创建线程2 */
40          tid2 = rt_thread_create("thread",
41                  thread_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
42                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
43          if (tid2 != RT_NULL)
44                  rt_thread_startup(tid2);
45          else
46                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
47
48          return 0;
49  }
50
51  #ifdef RT_USING_TC
52  static void _tc_cleanup()
53  {
54          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
55          rt_enter_critical();
56
57          /* 删除线程 */
58          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
59                  rt_thread_delete(tid1);
60          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
61                  rt_thread_delete(tid2);
62
63          /* 调度器解锁 */
64          rt_exit_critical();
65
66          /* 设置TestCase状态 */
67          tc_done(TC_STAT_PASSED);
68  }
69
70  int _tc_thread_dynamic_simple()
71  {
72          /* 设置TestCase清理回调函数 */
73          tc_cleanup(_tc_cleanup);
74          thread_dynamic_simple_init();
75
76          /* 返回TestCase运行的最长时间 */
77          return 100;
78  }
79  /* 输出函数命令到finsh shell中 */
80  FINSH_FUNCTION_EXPORT(_tc_thread_dynamic_simple, a dynamic thread example);
81  #else
82  /* 用户应用入口 */
83  int rt_application_init()
84  {
85          thread_dynamic_simple_init();
86
87          return 0;
88  }
```

```
#endif
```

6.7.2 线程删除

一个线程通过rt_thread_create创建出来后，因为出错或其他原因需要删除一个线程。当需要删除用rt_thread_create创建出的线程时，可以使用以下接口：

```
rt_err_t rt_thread_delete (rt_thread_t thread)
```

调用该接口后，线程对象将会被移出线程队列并且从内核对象管理器中删除，线程占用的堆栈空间也会被释放以回收相应的空间进行其他内存分配。

rt_thread_delete删除线程接口仅把相应的线程状态更改为RT_THREAD_STATE_CLOSE状态，然后放入到 `rt_thread_defunct` 队列中。真正的删除（释放线程控制块和释放线程栈）需要到下一次执行idle线程时，由idle线程完成最后的线程删除动作。用rt_thread_init初始化的静态线程请不要使用此接口删除。

Note: 线程运行完成，自动结束时，系统会自动删除线程，不需要再调用rt_thread_delete接口。这个接口*不应由*线程本身来调用以删除自身，一般由其他线程调用或在定时器超时函数中调用。

线程删除例子如下：

```
/*
 * 程序清单：删除线程
 *
 * 这个例子会创建两个线程，在一个线程中删除另外一个线程。
 */
#include <rtthread.h>
#include "tc_comm.h"

/*
 * 线程删除(rt_thread_delete)函数仅适合于动态线程，为了在一个线程
 * 中访问另一个线程的控制块，所以把线程块指针声明成全局类型以供全
 * 局访问
 */
static rt_thread_t tid1 = RT_NULL, tid2 = RT_NULL;
/* 线程1的入口函数 */
static void thread1_entry(void* parameter)
{
	rt_uint32_t count = 0;

	while (1)
	{
		/* 线程1采用低优先级运行，一直打印计数值 */
		rt_kprintf("thread count: %d\n", count ++);
	}
}

/* 线程2的入口函数 */
static void thread2_entry(void* parameter)
{
```

```
30          /* 线程2拥有较高的优先级，以抢占线程1而获得执行 */
31
32          /* 线程2启动后先睡眠10个OS Tick */
33          rt_thread_delay(10);
34
35          /*
36           * 线程2唤醒后直接删除线程1，删除线程1后，线程1自动脱离就绪线程
37           * 队列
38           */
39          rt_thread_delete(tid1);
40          tid1 = RT_NULL;
41
42          /*
43           * 线程2继续休眠10个OS Tick然后退出，线程2休眠后应切换到idle线程
44           * idle线程将执行真正的线程1控制块和线程栈的删除
45           */
46          rt_thread_delay(10);
47
48          /*
49           * 线程2运行结束后也将自动被删除(线程控制块和线程栈依然在idle线
50           * 程中释放)
51           */
52          tid2 = RT_NULL;
53  }
54
55  /* 线程删除示例的初始化 */
56  int thread_delete_init()
57  {
58          /* 创建线程1 */
59          tid1 = rt_thread_create("t1", /* 线程1的名称是t1 */
60                  thread1_entry, RT_NULL,   /* 入口是thread1_entry，参数是RT_NULL */
61                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
62          if (tid1 != RT_NULL) /* 如果获得线程控制块，启动这个线程 */
63                  rt_thread_startup(tid1);
64          else
65                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
66
67          /* 创建线程1 */
68          tid2 = rt_thread_create("t2", /* 线程1的名称是t2 */
69                  thread2_entry, RT_NULL,   /* 入口是thread2_entry，参数是RT_NULL */
70                  THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
71          if (tid2 != RT_NULL) /* 如果获得线程控制块，启动这个线程 */
72                  rt_thread_startup(tid2);
73          else
74                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
75
76          return 0;
77  }
78
79  #ifdef RT_USING_TC
80  static void _tc_cleanup()
81  {
82          /* lock scheduler */
```

```
83          rt_enter_critical();
84
85          /* delete thread */
86          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
87                  tc_stat(TC_STAT_FAILED);
88          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
89                  tc_stat(TC_STAT_FAILED);
90
91          /* unlock scheduler */
92          rt_exit_critical();
93  }
94
95  int _tc_thread_delete()
96  {
97          /* set tc cleanup */
98          tc_cleanup(_tc_cleanup);
99          thread_delete_init();
100
101         return 100;
102 }
103 FINSH_FUNCTION_EXPORT(_tc_thread_delete, a thread delete example);
104 #else
105 int rt_application_init()
106 {
107         thread_delete_init();
108
109         return 0;
110 }
111 #endif
```

6.7.3 线程初始化

线程的初始化可以使用以下接口完成：

```
rt_err_t rt_thread_init(struct rt_thread* thread,
        const char* name,
        void (*entry)(void* parameter), void* parameter,
        void* stack_start, rt_uint32_t stack_size,
        rt_uint8_t priority, rt_uint32_t tick);
```

通常线程初始化函数用来初始化静态线程对象，线程句柄，线程栈由用户提供，一般都设置为全局变量在编译时被分配，内核不负责动态分配空间。**时间片(tick)** 的单位是操作系统的时钟节拍，当系统中存在相同优先级线程时，这个参数指定一次这个线程能够运行最大的时间数，这段时间片运行结束后，调度器自动选择下一个就绪的同优先级线程进行运行。

线程初始化例子如下所示：

```
1   /*
2    * 程序清单：静态线程
3    *
4    * 这个程序会初始化2个静态线程，它们拥有共同的入口函数，但参数不相同
```

```
5    */
6   #include <rtthread.h>
7   #include "tc_comm.h"
8
9   /* 线程1控制块 */
10  static struct rt_thread thread1;
11  /* 线程1栈 */
12  static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];
13  /* 线程2控制块 */
14  static struct rt_thread thread2;
15  /* 线程2栈 */
16  static rt_uint8_t thread2_stack[THREAD_STACK_SIZE];
17
18  /* 线程入口 */
19  static void thread_entry(void* parameter)
20  {
21          rt_uint32_t count = 0;
22          rt_uint32_t no = (rt_uint32_t) parameter; /* 获得正确的入口参数 */
23
24          while (1)
25          {
26                  /* 打印线程计数值输出 */
27                  rt_kprintf("thread%d count: %d\n", no, count ++);
28
29                  /* 休眠10个OS Tick */
30                  rt_thread_delay(10);
31          }
32  }
33
34  int thread_static_simple_init()
35  {
36          rt_err_t result;
37
38          /* 初始化线程1 */
39          result = rt_thread_init(&thread1, "t1", /* 线程名：t1 */
40                  thread_entry, (void*)1, /* 线程的入口是thread_entry，入口参数是1 */
41                  &thread1_stack[0], sizeof(thread1_stack), /* 线程栈是thread1_stack */
42                  THREAD_PRIORITY, 10);
43          if (result == RT_EOK) /* 如果返回正确，启动线程1 */
44                  rt_thread_startup(&thread1);
45          else
46                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
47
48          /* 初始化线程2 */
49          result = rt_thread_init(&thread2, "t2", /* 线程名：t2 */
50                  thread_entry, RT_NULL, /* 线程的入口是thread_entry，入口参数是2 */
51                  &thread2_stack[0], sizeof(thread2_stack), /* 线程栈是thread2_stack */
52                  THREAD_PRIORITY + 1, 10);
53          if (result == RT_EOK) /* 如果返回正确，启动线程2 */
54                  rt_thread_startup(&thread2);
55          else
56                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
57
```

```
58          return 0;
59  }
60
61  #ifdef RT_USING_TC
62  static void _tc_cleanup()
63  {
64          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
65          rt_enter_critical();
66
67          /* 执行线程脱离 */
68          if (thread1.stat != RT_THREAD_CLOSE)
69                  rt_thread_detach(&thread1);
70          if (thread2.stat != RT_THREAD_CLOSE)
71                  rt_thread_detach(&thread2);
72
73          /* 调度器解锁 */
74          rt_exit_critical();
75
76          /* 设置TestCase状态 */
77          tc_done(TC_STAT_PASSED);
78  }
79
80  int _tc_thread_static_simple()
81  {
82          /* 设置TestCase清理回调函数 */
83          tc_cleanup(_tc_cleanup);
84          thread_static_simple_init();
85
86          /* 返回TestCase运行的最长时间 */
87          return 100;
88  }
89  /* 输出函数命令到finsh shell中 */
90  FINSH_FUNCTION_EXPORT(_tc_thread_static_simple, a static thread example);
91  #else
92  /* 用户应用入口 */
93  int rt_application_init()
94  {
95          thread_static_simple_init();
96
97          return 0;
98  }
99  #endif
```

6.7.4 线程脱离

脱离线程将使线程对象被从线程队列和内核对象管理器中删除。脱离线程使用以下接口。

```
rt_err_t rt_thread_detach (rt_thread_t thread)
```

注：这个函数接口是和rt_thread_delete相对应的，rt_thread_delete操作的对象是rt_thread_create创建的句柄，而rt_thread_detach操作的对象是使用rt_thread_init初始化的线程控制块。同样，线程本身不应调用这个接口脱离线程本身。

线程脱离的例子如下所示：

```
/*
 * 程序清单：线程脱离
 *
 * 这个例子会创建两个线程，在其中一个线程中执行对另一个线程的脱离。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 线程1控制块 */
static struct rt_thread thread1;
/* 线程1栈 */
static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];
/* 线程2控制块 */
static struct rt_thread thread2;
/* 线程2栈 */
static rt_uint8_t thread2_stack[THREAD_STACK_SIZE];

/* 线程1入口 */
static void thread1_entry(void* parameter)
{
        rt_uint32_t count = 0;

        while (1)
        {
                /* 线程1采用低优先级运行，一直打印计数值 */
                rt_kprintf("thread count: %d\n", count ++);
        }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
        /* 线程2拥有较高的优先级，以抢占线程1而获得执行 */

        /* 线程2启动后先睡眠10个OS Tick */
        rt_thread_delay(10);

        /*
         * 线程2唤醒后直接执行线程1脱离，线程1将从就绪线程队列中删除
         */
        rt_thread_detach(&thread1);

        /*
         * 线程2继续休眠10个OS Tick然后退出
         */
        rt_thread_delay(10);

        /*
         * 线程2运行结束后也将自动被从就绪队列中删除，并脱离线程队列
         */
}
```

```
52
53  int thread_detach_init()
54  {
55          rt_err_t result;
56
57          /* 初始化线程1 */
58          result = rt_thread_init(&thread1, "t1", /* 线程名: t1 */
59                  thread1_entry, RT_NULL, /* 线程的入口是thread1_entry, 入口参数是RT_NULL*/
60                  &thread1_stack[0], sizeof(thread1_stack), /* 线程栈是thread1_stack */
61                  THREAD_PRIORITY, 10);
62          if (result == RT_EOK) /* 如果返回正确, 启动线程1 */
63                  rt_thread_startup(&thread1);
64          else
65                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
66
67          /* 初始化线程2 */
68          result = rt_thread_init(&thread2, "t2", /* 线程名: t2 */
69                  thread2_entry, RT_NULL, /* 线程的入口是thread2_entry, 入口参数是RT_NULL*/
70                  &thread2_stack[0], sizeof(thread2_stack), /* 线程栈是thread2_stack */
71                  THREAD_PRIORITY - 1, 10);
72          if (result == RT_EOK) /* 如果返回正确, 启动线程2 */
73                  rt_thread_startup(&thread2);
74          else
75                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
76
77          return 0;
78  }
79
80  #ifdef RT_USING_TC
81  static void _tc_cleanup()
82  {
83          /* 调度器上锁, 上锁后, 将不再切换到其他线程, 仅响应中断 */
84          rt_enter_critical();
85
86          /* 执行线程脱离 */
87          if (thread1.stat != RT_THREAD_CLOSE)
88                  rt_thread_detach(&thread1);
89          if (thread2.stat != RT_THREAD_CLOSE)
90                  rt_thread_detach(&thread2);
91
92          /* 调度器解锁 */
93          rt_exit_critical();
94
95          /* 设置TestCase状态 */
96          tc_done(TC_STAT_PASSED);
97  }
98
99  int _tc_thread_detach()
100 {
101         /* 设置TestCase清理回调函数 */
102         tc_cleanup(_tc_cleanup);
103         thread_detach_init();
104
```

```
105         /* 返回TestCase运行的最长时间 */
106         return 100;
107 }
108 /* 输出函数命令到finsh shell中 */
109 FINSH_FUNCTION_EXPORT(_tc_thread_detach, a static thread example);
110 #else
111 /* 用户应用入口 */
112 int rt_application_init()
113 {
114         thread_detach_init();
115 
116         return 0;
117 }
118 #endif
```

6.7.5 线程启动

创建/初始化的线程对象的状态是RT_THREAD_INIT状态，并未进入调度队列，可以调用如下接口启动一个创建/初始化的线程对象：

```
rt_err_t rt_thread_startup (rt_thread_t thread)
```

6.7.6 当前线程

在程序的运行过程中，相同的一段代码可能会被多个线程执行，在执行的时候可以通过下面的接口获得当前执行的线程句柄。

```
rt_thread_t rt_thread_self (void)
```

Note: 请不要在中断服务程序中调用此函数，因为它并不能准确获得当前的执行线程。当调度器未启动时，这个接口返回RT_NULL。

6.7.7 线程让出处理机

当前线程的时间片用完或者该线程自动要求让出处理器资源时，它不再占有处理机，调度器会选择下一个最高优先级的线程执行。这时，放弃处理器资源的线程仍然在就绪队列中。线程让出处理器使用以下接口：

```
rt_err_t rt_thread_yield ()
```

调用该接口后，线程首先把自己从它所在的队列中删除，然后把自己挂到与该线程优先级对应的就绪线程链表的尾部，然后激活调度器切换到优先级最高的线程。

线程让出处理机代码例子如下所示：

```
1 /*
2  * 程序清单：
3  */
4 #include <rtthread.h>
```

```
5  #include "tc_comm.h"
6
7  /* 指向线程控制块的指针 */
8  static rt_thread_t tid1 = RT_NULL;
9  static rt_thread_t tid2 = RT_NULL;
10 /* 线程1入口 */
11 static void thread1_entry(void* parameter)
12 {
13         rt_uint32_t count = 0;
14
15         while (1)
16         {
17                 /* 打印线程1的输出 */
18                 rt_kprintf("thread1: count = %d\n", count ++);
19
20                 /* 执行yield后应该切换到thread2执行 */
21                 rt_thread_yield();
22         }
23 }
24
25 /* 线程2入口 */
26 static void thread2_entry(void* parameter)
27 {
28         rt_uint32_t count = 0;
29
30         while (1)
31         {
32                 /* 打印线程2的输出 */
33                 rt_kprintf("thread2: count = %d\n", count ++);
34
35                 /* 执行yield后应该切换到thread1执行 */
36                 rt_thread_yield();
37         }
38 }
39
40 int thread_yield_init()
41 {
42         /* 创建线程1 */
43         tid1 = rt_thread_create("thread",
44                 thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
45                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
46         if (tid1 != RT_NULL)
47                 rt_thread_startup(tid1);
48         else
49                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
50
51         /* 创建线程2 */
52         tid2 = rt_thread_create("thread",
53                 thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
54                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
55         if (tid2 != RT_NULL)
56                 rt_thread_startup(tid2);
57         else
```

```
58                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
59
60          return 0;
61  }
62
63  #ifdef RT_USING_TC
64  static void _tc_cleanup()
65  {
66          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
67          rt_enter_critical();
68
69          /* 删除线程 */
70          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
71                  rt_thread_delete(tid1);
72          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
73                  rt_thread_delete(tid2);
74
75          /* 调度器解锁 */
76          rt_exit_critical();
77
78          /* 设置TestCase状态 */
79          tc_done(TC_STAT_PASSED);
80  }
81
82  int _tc_thread_yield()
83  {
84          /* 设置TestCase清理回调函数 */
85          tc_cleanup(_tc_cleanup);
86          thread_yield_init();
87
88          /* 返回TestCase运行的最长时间 */
89          return 100;
90  }
91  /* 输出函数命令到finsh shell中 */
92  FINSH_FUNCTION_EXPORT(_tc_thread_yield, a thread yield example);
93  #else
94  /* 用户应用入口 */
95  int rt_application_init()
96  {
97          thread_yield_init();
98
99          return 0;
100 }
101 #endif
```

Note: rt_thread_yield函数和rt_schedule函数比较相像，但在相同优先级线程存在时，系统的行为是完全不一样的。当有相同优先级就绪线程存在，并且系统中不存在更高优先级的就绪线程时，执行rt_thread_yield函数后，当前线程被换出。而执行rt_schedule函数后，当前线程并不被换成。

6.7.8 线程睡眠

在实际应用中，经常需要线程延迟一段时间，指定的时间到达后，线程重新运行，线程睡眠使用以下接口：

```
rt_err_t rt_thread_sleep(rt_tick_t tick)
rt_err_t rt_thread_delay(rt_tick_t tick)
```

以上两个接口作用是相同的，调用该线程可以使线程暂时挂起指定的时间，它接受一个参数，该参数指定了线程的休眠时间（OS Tick时钟节拍数）。

6.7.9 线程挂起

当线程调用rt_thread_delay，rt_sem_take，rt_mb_recv等函数时，将主动挂起。或者由于线程获取不到资源，它也会进入到挂起状态。在挂起状态的线程，如果等待的资源超时或由于其他线程释放资源，它将返回到就绪状态。挂起线程使用以下接口：

```
rt_err_t rt_thread_suspend (rt_thread_t thread)
```

注：如果挂起当前任务，需要在调用这个函数后，紧接着调用rt_schedule函数进行*手动的线程上下文切换*。

挂起线程的代码例子如下所示。

```
1  /*
2   * 程序清单：挂起线程
3   *
4   * 这个例子中将创建两个动态线程，高优先级线程将在一定时刻后挂起低优先级线程。
5   */
6  #include <rtthread.h>
7  #include "tc_comm.h"
8
9  /* 指向线程控制块的指针 */
10 static rt_thread_t tid1 = RT_NULL;
11 static rt_thread_t tid2 = RT_NULL;
12 /* 线程1入口 */
13 static void thread1_entry(void* parameter)
14 {
15         rt_uint32_t count = 0;
16
17         while (1)
18         {
19                 /* 线程1采用低优先级运行，一直打印计数值 */
20                 rt_kprintf("thread count: %d\n", count ++);
21         }
22 }
23
24 /* 线程2入口 */
25 static void thread2_entry(void* parameter)
26 {
27         /* 延时10个OS Tick */
28         rt_thread_delay(10);
```

```
29
30          /* 挂起线程1 */
31          rt_thread_suspend(tid1);
32
33          /* 延时10个OS Tick */
34          rt_thread_delay(10);
35
36          /* 线程2自动退出 */
37  }
38
39  int thread_suspend_init()
40  {
41          /* 创建线程1 */
42          tid1 = rt_thread_create("thread",
43                  thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
44                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
45          if (tid1 != RT_NULL)
46                  rt_thread_startup(tid1);
47          else
48                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
49
50          /* 创建线程2 */
51          tid2 = rt_thread_create("thread",
52                  thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
53                  THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
54          if (tid2 != RT_NULL)
55                  rt_thread_startup(tid2);
56          else
57                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
58
59          return 0;
60  }
61
62  #ifdef RT_USING_TC
63  static void _tc_cleanup()
64  {
65          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
66          rt_enter_critical();
67
68          /* 删除线程 */
69          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
70                  rt_thread_delete(tid1);
71          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
72                  rt_thread_delete(tid2);
73
74          /* 调度器解锁 */
75          rt_exit_critical();
76
77          /* 设置TestCase状态 */
78          tc_done(TC_STAT_PASSED);
79  }
80
81  int _tc_thread_suspend()
```

```
{
        /* 设置TestCase清理回调函数 */
        tc_cleanup(_tc_cleanup);
        thread_suspend_init();

        /* 返回TestCase运行的最长时间 */
        return 100;
}
/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_thread_suspend, a thread suspend example);
#else
/* 用户应用入口 */
int rt_application_init()
{
        thread_suspend_init();

        return 0;
}
#endif
```

6.7.10 线程恢复

线程恢复使得挂起的线程重新进入就绪状态。线程恢复使用以下接口：

```
rt_err_t rt_thread_resume (rt_thread_t thread)
```

恢复挂起线程的代码例子如下所示。

```
/*
 * 程序清单：唤醒线程
 *
 * 这个例子中将创建两个动态线程，低优先级线程将挂起自身，然后
 * 高优先级线程将在一定时刻后唤醒低优先级线程。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程1入口 */
static void thread1_entry(void* parameter)
{
        /* 低优先级线程1开始运行 */
        rt_kprintf("thread1 startup%d\n");

        /* 挂起自身 */
        rt_kprintf("suspend thread self\n");
        rt_thread_suspend(tid1);
        /* 主动执行线程调度 */
        rt_schedule();

```

```
25          /* 当线程1被唤醒时 */
26          rt_kprintf("thread1 resumed\n");
27  }
28
29  /* 线程2入口 */
30  static void thread2_entry(void* parameter)
31  {
32          /* 延时10个OS Tick */
33          rt_thread_delay(10);
34
35          /* 唤醒线程1 */
36          rt_thread_resume(tid1);
37
38          /* 延时10个OS Tick */
39          rt_thread_delay(10);
40
41          /* 线程2自动退出 */
42  }
43
44  int thread_resume_init()
45  {
46          /* 创建线程1 */
47          tid1 = rt_thread_create("thread",
48                  thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
49                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
50          if (tid1 != RT_NULL)
51                  rt_thread_startup(tid1);
52          else
53                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
54
55          /* 创建线程2 */
56          tid2 = rt_thread_create("thread",
57                  thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
58                  THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
59          if (tid2 != RT_NULL)
60                  rt_thread_startup(tid2);
61          else
62                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
63
64          return 0;
65  }
66
67  #ifdef RT_USING_TC
68  static void _tc_cleanup()
69  {
70          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
71          rt_enter_critical();
72
73          /* 删除线程 */
74          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
75                  rt_thread_delete(tid1);
76          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
77                  rt_thread_delete(tid2);
```

```
78
79          /* 调度器解锁 */
80          rt_exit_critical();
81
82          /* 设置TestCase状态 */
83          tc_done(TC_STAT_PASSED);
84  }
85
86  int _tc_thread_resume()
87  {
88          /* 设置TestCase清理回调函数 */
89          tc_cleanup(_tc_cleanup);
90          thread_resume_init();
91
92          /* 返回TestCase运行的最长时间 */
93          return 100;
94  }
95  /* 输出函数命令到finsh shell中 */
96  FINSH_FUNCTION_EXPORT(_tc_thread_resume, a thread resume example);
97  #else
98  /* 用户应用入口 */
99  int rt_application_init()
100 {
101         thread_resume_init();
102
103         return 0;
104 }
105 #endif
```

6.7.11 线程控制

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下接口：

```
rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg)
```

cmd指示出控制命令，当前支持的命令包括：

- RT_THREAD_CTRL_CHANGE_PRIORITY - 动态更改线程的优先级
- RT_THREAD_CTRL_STARTUP - 开始运行一个线程
- RT_THREAD_CTRL_CLOSE - 关闭一个线程，等同于rt_thread_delete。

arg必须是一个线程控制块指针。

6.7.12 初始化空闲线程

在系统调度器运行前，必须通过调用如下的函数初始化空闲线程。

```
void rt_thread_idle_init(void)
```

6.7.13 设置空闲线程钩子

可以调用如下的函数，设置空闲线程运行时执行的钩子函数。

```
void rt_thread_idle_set_hook(void (*hook)())
```

当空闲线程运行时会自动执行设置的钩子函数，由于空闲线程具有系统的最低优先级，所以只有在空闲时刻才会执行此钩子函数。空闲线程是一个线程状态永远为就绪状态的线程，因此挂入的钩子函数必须保证空闲线程永远不会处于挂起状态，例如rt_thread_delay，rt_sem_take等可能会导致线程挂起的函数不能使用。

CHAPTER

SEVEN

线程间同步与通信

在多任务实时系统中，一项工作的完成往往可以通过多个任务协调方式来共同完成，例如一个任务从数据采集器中读取数据，然后放到一个链表中进行保存。而另一个任务则从这个链表队列中把数据取出来进行分析处理，并把数据从链表中删除（一个典型的消费者与生产者的例子）。

当消费者任务取到链表的最末端的时候，此时生产者任务可能正在往末端添加数据，那么就很有可能生产者拿到的末节点被消费者任务给删除了，从而整个程序的运行逻辑被打乱，运行结果也将不可预测。

正常的操作顺序应该是在一个任务删除或添加动作完成时再进行下一个动作，生产任务与消费任务之间需要协调动作：即某些单一的动作必须保证它们只能被一个任务完整操作完成。对于操作/访问同一块区域，称之为临界区。任务的同步方式有很多中，其核心思想都是：在访问临界区的时候只允许一个任务运行。

7.1 关闭中断

关闭中断是禁止多任务访问临界区最简单的一种方式，即使是在分时操作系统中也是如此。当关闭中断的时候，就意味着当前任务不会被其他事件所打断（因为整个系统已经不再响应外部事件，如果自己不主动放弃处理机，它也不会产生内部事件），也就是当前任务不会被抢占，除非这个任务主动放弃了处理机。关闭中断/恢复中断 API接口是由BSP实现的，根据不同的平台其实现方式也不大相同。

关闭、打开中断由两个函数完成：

- 关闭中断

  ```
  rt_base_t rt_hw_interrupt_disable(void);
  ```

  关闭中断并返回关闭中断前的中断状态

- 恢复中断

  ```
  void rt_hw_interrupt_enable(rt_base_t level);
  ```

  使能中断，它采用恢复调用rt_hw_interrupt_disable前的中断状态进行恢复中断状态，如果调用rt_hw_interrupt_disable（）前是关中断状态，那么调用此函数后依然是关中断状态。

使用的例子代码如下：

```c
/* 代码清单：关闭中断进行全局变量的访问 */
#include <rtthread.h>

/* 同时访问的全局变量 */
static rt_uint32_t cnt;
void thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t level;

    no = (rt_uint32_t) parameter;
    while(1)
    {
        /* 关闭中断 */
        level = rt_hw_interrupt_disable();
        cnt += no;
        /* 恢复中断 */
        rt_hw_interrupt_enable(level);

        rt_kprintf("thread[%d]'s counter is %d\n", no, cnt);
        rt_thread_delay(no);
    }
}

/* 用户应用程序入口 */
void rt_application_init()
{
    rt_thread_t thread;

    /* 创建t10线程 */
    thread = rt_thread_create("t10", thread_entry, (void*)10,
        512, 10, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);

    /* 创建t20线程 */
    thread = rt_thread_create("t20", thread_entry, (void*)20,
        512, 20, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);
}
```

Warning: 由于关闭中断会导致整个系统不能响应外部中断，所以在使用关闭中断做为互斥访问临界区的手段时，首先需要保证的是关闭中断的时间非常短，例如数条机器指令。

7.2 调度器上锁

同样把调度器锁住也能让当前运行的任务不被换出，直到调度器解锁。但和关闭中断有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然有可能被运行。所以在使用调度器上锁的方式来做任务同步时，需要考虑好，任务访问的临界资源是否会被中断服务例程所修改，如果可能会被修改，那么将不适合采用此种方式作为同步的方法。

RT-Thread提供的调度器操作API为

```
void rt_enter_critical(void); /* 进入临界区 */
```

调用这个函数后，调度器将被上锁。在系统锁住调度器的期间，系统依然响应中断，但因为中断而可能唤醒了高优先级的任务，调度器依然不会选择高优先级的任务执行，直到调用解锁调度器函数才尝试进行下一次调度。

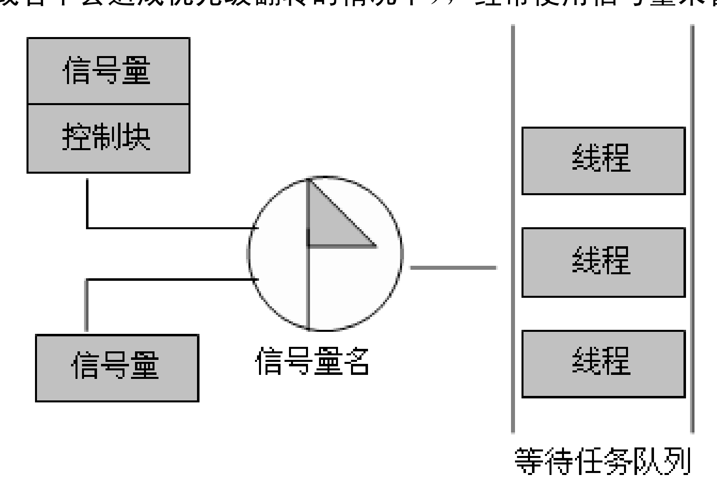
```
void rt_exit_critical(void); /* 退出临界区 */
```

在系统退出临界区的时候，系统会计算当前是否有更高优先级的任务就绪，如果有比当前线程更高优先级的线程就绪，那么将切换到这个高优先级线程中执行；如果无更高优先级线程就绪，那么将继续执行当前任务。

Note: rt_enter_critical/rt_exit_critical可以多次嵌套调用，但有多少次rt_enter_critical就必须有成对的rt_exit_critical调用，嵌套的最大深度是255。

7.3 信号量

信号量是用来解决线程同步和互斥的通用工具，和互斥量类似，信号量也可用作资源互斥访问，但信号量没有所有者的概念，在应用上比互斥量更广泛。信号量比较简单，不能解决优先级翻转问题，但信号量是一种轻量级的对象，比互斥量小巧、灵活。因此在很多对互斥要求不严格的系统中（或者不会造成优先级翻转的情况下），经常使用信号量来管理互斥资源。



信号量工作示意图如上图所示，每个信号量对象有一个信号量值和一个线程等待队列，信号量的值对应了信号量对象的实例数目，假如信号量值为5，则表示共有5个信号量实例可以被使用，当信号量实例数目为零时，再申请该信号量的对象就会被挂起在该信号量的等待队列上，等待可用的信号量实例。

7.3.1 信号量控制块

```
struct rt_semaphore
{
    struct rt_ipc_object parent;     /* 信号量对象继承自ipc对象 */
```

```
    rt_base_t value;                /* 信号量的值                */
};
```

rt_semaphore对象从rt_ipc_object中派生，由IPC容器所管理。

7.3.2 信号量相关接口

创建信号量

当创建一个信号量时，内核首先创建一个信号量控制块，然后对该控制块进行基本的初始化工作，创建信号量使用以下接口：

```
rt_sem_t rt_sem_create (const char* name, rt_uint32_t value, rt_uint8_t flag);
```

使用该接口时，需为信号量指定一个名称，并指定信号量初始值和信号量标志。信号量标志可以是基于优先级方式或基于FIFO方式：

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式      */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式     */
```

创建信号量的例程如下程序清单所示：

```
1  /*
2   * 程序清单：动态信号量
3   *
4   * 这个例子中将创建一个动态信号量（初始值为0 ）及一个动态线程，在这个动态线程中
5   * 将试图采用超时方式去持有信号量，应该超时返回。然后这个线程释放一次信号量，并
6   * 在后面继续采用永久等待方式去持有信号量，  成功获得信号量后返回。
7   */
8  #include <rtthread.h>
9  #include "tc_comm.h"
10
11 /* 指向线程控制块的指针 */
12 static rt_thread_t tid = RT_NULL;
13 /* 指向信号量的指针 */
14 static rt_sem_t sem = RT_NULL;
15 /* 线程入口 */
16 static void thread_entry(void* parameter)
17 {
18         rt_err_t result;
19         rt_tick_t tick;
20
21         /* 获得当前的OS Tick */
22         tick = rt_tick_get();
23
24         /* 视图持有一个信号量，如果10个OS Tick依然没拿到，则超时返回 */
25         result = rt_sem_take(sem, 10);
26         if (result == -RT_ETIMEOUT)
27         {
28                 /* 判断是否刚好过去10个OS Tick */
29                 if (rt_tick_get() - tick != 10)
```

```
30                  {
31                          /* 如果失败，则测试失败 */
32                          tc_done(TC_STAT_FAILED);
33                          rt_sem_delete(sem);
34                          return;
35                  }
36                  rt_kprintf("take semaphore timeout\n");
37          }
38          else
39          {
40                  /* 因为并没释放信号量，应该是超时返回，否则测试失败 */
41                  tc_done(TC_STAT_FAILED);
42                  rt_sem_delete(sem);
43                  return;
44          }
45
46          /* 释放一次信号量 */
47          rt_sem_release(sem);
48
49          /* 继续持有信号量，并永远等待直到持有到信号量 */
50          result = rt_sem_take(sem, RT_WAITING_FOREVER);
51          if (result != RT_EOK)
52          {
53                  /* 返回不正确，测试失败 */
54                  tc_done(TC_STAT_FAILED);
55                  rt_sem_delete(sem);
56                  return;
57          }
58
59          /* 测试成功 */
60          tc_done(TC_STAT_PASSED);
61          /* 删除信号量 */
62          rt_sem_delete(sem);
63  }
64
65  int semaphore_dynamic_init()
66  {
67          /* 创建一个信号量，初始值是0 */
68          sem = rt_sem_create("sem", 0, RT_IPC_FLAG_FIFO);
69          if (sem == RT_NULL)
70          {
71                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
72                  return 0;
73          }
74
75          /* 创建线程 */
76          tid = rt_thread_create("thread",
77                  thread_entry, RT_NULL, /* 线程入口是thread_entry, 入口参数是RT_NULL */
78                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
79          if (tid != RT_NULL)
80                  rt_thread_startup(tid);
81          else
82                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
```

```
83
84          return 0;
85  }
86
87  #ifdef RT_USING_TC
88  static void _tc_cleanup()
89  {
90          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
91          rt_enter_critical();
92
93          /* 删除线程 */
94          if (tid != RT_NULL && tid->stat != RT_THREAD_CLOSE)
95          {
96                  rt_thread_delete(tid);
97
98                  /* 删除信号量 */
99                  rt_sem_delete(sem);
100         }
101
102         /* 调度器解锁 */
103         rt_exit_critical();
104
105         /* 设置TestCase状态 */
106         tc_done(TC_STAT_PASSED);
107 }
108
109 int _tc_semaphore_dynamic()
110 {
111         /* 设置TestCase清理回调函数 */
112         tc_cleanup(_tc_cleanup);
113         semaphore_dynamic_init();
114
115         /* 返回TestCase运行的最长时间 */
116         return 100;
117 }
118 /* 输出函数命令到finsh shell中 */
119 FINSH_FUNCTION_EXPORT(_tc_semaphore_dynamic, a dynamic semaphore example);
120 #else
121 /* 用户应用入口 */
122 int rt_application_init()
123 {
124         semaphore_dynamic_init();
125
126         return 0;
127 }
128 #endif
```

删除信号量

系统不再使用信号量时，通过删除信号量以释放系统资源。删除信号量使用以下接口：

```
rt_err_t rt_sem_delete (rt_sem_t sem);
```

删除一个信号量，必须确保该信号量不再被使用。如果删除该信号量时，有线程正在等待该信号量，则先唤醒等待在该信号量上的线程（返回值为-RT_ERROR）。

初始化信号量

对于静态信号量对象，它的内存空间在编译时期就被编译器分配出来，放在数据段或ZI段上，此时使用信号量就不再需要使用rt_sem_create接口来创建它，而只需在使用前对它进行初始化即可。初始化信号量对象可使用以下接口：

```
rt_err_t rt_sem_init (rt_sem_t sem, const char* name, rt_uint32_t value, rt_uint8_t flag)
```

使用该接口时，需指定给出信号量对象的句柄（指向信号量控制块的指针），指定信号量名称，信号量初始值以及信号量标志参数。

相应例程如下面代码清单中所示：

```
1  /*
2   * 程序清单：静态信号量
3   *
4   * 这个例子中将创建一个静态信号量（初始值为0 ）及一个静态线程，在这个静态线程中
5   * 将试图采用超时方式去持有信号量，应该超时返回。然后这个线程释放一次信号量，并
6   * 在后面继续采用永久等待方式去持有信号量， 成功获得信号量后返回。
7   */
8  #include <rtthread.h>
9  #include "tc_comm.h"
10
11 /* 线程控制块及栈 */
12 static struct rt_thread thread;
13 static rt_uint8_t thread_stack[THREAD_STACK_SIZE];
14 /* 信号量控制块 */
15 static struct rt_semaphore sem;
16
17 /* 线程入口 */
18 static void thread_entry(void* parameter)
19 {
20         rt_err_t result;
21         rt_tick_t tick;
22
23         /* 获得当前的OS Tick */
24         tick = rt_tick_get();
25
26         /* 试图持有信号量，最大等待10个OS Tick后返回 */
27         result = rt_sem_take(&sem, 10);
28         if (result == -RT_ETIMEOUT)
29         {
30                 /* 超时后判断是否刚好是10个OS Tick */
31                 if (rt_tick_get() - tick != 10)
32                 {
33                         tc_done(TC_STAT_FAILED);
```

```
34                          rt_sem_detach(&sem);
35                          return;
36                  }
37                  rt_kprintf("take semaphore timeout\n");
38          }
39          else
40          {
41                  /* 因为没有其他地方是否信号量，所以不应该成功持有信号量，否则测试失败 */
42                  tc_done(TC_STAT_FAILED);
43                  rt_sem_detach(&sem);
44                  return;
45          }
46
47          /* 释放一次信号量 */
48          rt_sem_release(&sem);
49
50          /* 永久等待方式持有信号量 */
51          result = rt_sem_take(&sem, RT_WAITING_FOREVER);
52          if (result != RT_EOK)
53          {
54                  /* 不成功则测试失败 */
55                  tc_done(TC_STAT_FAILED);
56                  rt_sem_detach(&sem);
57                  return;
58          }
59
60          /* 测试通过 */
61          tc_done(TC_STAT_PASSED);
62          /* 脱离信号量对象 */
63          rt_sem_detach(&sem);
64  }
65
66  int semaphore_static_init()
67  {
68          rt_err_t result;
69
70          /* 初始化信号量，初始值是0 */
71          result = rt_sem_init(&sem, "sem", 0, RT_IPC_FLAG_FIFO);
72          if (result != RT_EOK)
73          {
74                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
75                  return 0;
76          }
77
78          /* 初始化线程1 */
79          result = rt_thread_init(&thread, "thread", /* 线程名：thread */
80                  thread_entry, RT_NULL, /* 线程的入口是thread_entry，入口参数是RT_NULL*/
81                  &thread_stack[0], sizeof(thread_stack), /* 线程栈是thread_stack */
82                  THREAD_PRIORITY, 10);
83          if (result == RT_EOK) /* 如果返回正确，启动线程1 */
84                  rt_thread_startup(&thread);
85          else
86                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
```

```
87
88          return 0;
89  }
90
91  #ifdef RT_USING_TC
92  static void _tc_cleanup()
93  {
94          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
95          rt_enter_critical();
96
97          /* 执行线程脱离 */
98          if (thread.stat != RT_THREAD_CLOSE)
99          {
100                 rt_thread_detach(&thread);
101
102                 /* 执行信号量对象脱离 */
103                 rt_sem_detach(&sem);
104         }
105
106         /* 调度器解锁 */
107         rt_exit_critical();
108
109         /* 设置TestCase状态 */
110         tc_done(TC_STAT_PASSED);
111 }
112
113 int _tc_semaphore_static()
114 {
115         /* 设置TestCase清理回调函数 */
116         tc_cleanup(_tc_cleanup);
117         semaphore_static_init();
118
119         /* 返回TestCase运行的最长时间 */
120         return 100;
121 }
122 /* 输出函数命令到finsh shell中 */
123 FINSH_FUNCTION_EXPORT(_tc_semaphore_static, a static semaphore example);
124 #else
125 /* 用户应用入口 */
126 int rt_application_init()
127 {
128         thread_static_init();
129
130         return 0;
131 }
132 #endif
```

脱离信号量

脱离信号量将把信号量对象从内核对象管理器中移除掉。脱离信号量使用以下接口。

```
rt_err_t rt_sem_detach (rt_sem_t sem)
```

使用该接口后，内核先唤醒所有挂在该信号量等待队列上的线程，然后将该信号量从内核对象管理器中删除。原挂起在信号量上的线程将获得-RT_ERROR的返回值。

获取信号量

线程通过获取信号量来获得信号量资源实例，当信号量值大于零时，它每次被申请获得，值都会减一，获取信号量使用以下接口：

```
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32_t time)
```

如果信号量的值等于零，那么说明当前资源不可用，申请该信号量的线程将根据time参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程释放该信号量。如果采用超时挂起的方式，如果在指定的时间内依然得不到信号量，线程将超时返回，返回值是-RT_ETIMEOUT。

获取无等待信号量

当用户不想在申请的信号量上挂起线程进行等待时，可以使用无等待信号量，获取无等待信号量使用以下接口：

```
rt_err_t rt_sem_trytake(rt_sem_t sem)
```

这个接口与rt_sem_take(sem, 0)的作用相同，即当线程申请的信号量资源不可用的时候，它不是等待在该信号量上，而是直接返回-RT_ETIMEOUT。

释放信号量

当线程完成信号量资源的访问后，应尽快释放它占据的信号量，使得其他线程能获得该信号量。释放信号量使用以下接口：

```
rt_err_t rt_sem_release(rt_sem_t sem)
```

当信号量的值等于零时，信号量值加一，并且唤醒等待在该信号量上的线程队列中的第一个线程，由它获取信号量。

使用信号量的例子

```
1  /*
2   * 程序清单：信号量实现生产者消费者间的互斥
3   *
4   * 在这个程序中，会创建两个线程，一个是生成者线程worker一个是消费者线程thread
5   *
6   * 在数据信息生产、消费的过程中，worker负责把数据将写入到环形buffer中，而thread
7   * 则从环形buffer中读出。
8   */
9  #include <rtthread.h>
10 #include "tc_comm.h"
11
```

```
12 /* 一个环形buffer的实现 */
13 struct rb
14 {
15         rt_uint16_t read_index, write_index;
16         rt_uint8_t *buffer_ptr;
17         rt_uint16_t buffer_size;
18 };
19 
20 /* 指向信号量控制块的指针 */
21 static rt_sem_t sem = RT_NULL;
22 /* 指向线程控制块的指针 */
23 static rt_thread_t tid = RT_NULL, worker = RT_NULL;
24 
25 /* 环形buffer的内存块（用数组体现出来） */
26 #define BUFFER_SIZE                256
27 #define BUFFER_ITEM                32
28 static rt_uint8_t working_buffer[BUFFER_SIZE];
29 struct rb working_rb;
30 
31 /* 初始化环形buffer，size指的是buffer的大小。注：这里并没对数据地址对齐做处理 */
32 static void rb_init(struct rb* rb, rt_uint8_t *pool, rt_uint16_t size)
33 {
34         RT_ASSERT(rb != RT_NULL);
35 
36         /* 对读写指针清零*/
37         rb->read_index = rb->write_index = 0;
38 
39         /* 设置环形buffer的内存数据块 */
40         rb->buffer_ptr = pool;
41         rb->buffer_size = size;
42 }
43 
44 /* 向环形buffer中写入数据 */
45 static rt_bool_t rb_put(struct rb* rb, const rt_uint8_t *ptr, rt_uint16_t length)
46 {
47         rt_size_t size;
48 
49         /* 判断是否有足够的剩余空间 */
50         if (rb->read_index > rb->write_index)
51                 size = rb->read_index - rb->write_index;
52         else
53                 size = rb->buffer_size - rb->write_index + rb->read_index;
54 
55         /* 没有多余的空间 */
56         if (size < length) return RT_FALSE;
57 
58         if (rb->read_index > rb->write_index)
59         {
60                 /* read_index - write_index 即为总的空余空间 */
61                 memcpy(&rb->buffer_ptr[rb->write_index], ptr, length);
62                 rb->write_index += length;
63         }
64         else
```

```
65          {
66                  if (rb->buffer_size - rb->write_index > length)
67                  {
68                          /* write_index 后面剩余的空间有足够的长度 */
69                          memcpy(&rb->buffer_ptr[rb->write_index], ptr, length);
70                          rb->write_index += length;
71                  }
72                  else
73                  {
74                          /*
75                           * write_index 后面剩余的空间不存在足够的长度，需要把部分数据复制到
76                           * 前面的剩余空间中
77                           */
78                          memcpy(&rb->buffer_ptr[rb->write_index], ptr,
79                                  rb->buffer_size - rb->write_index);
80                          memcpy(&rb->buffer_ptr[0], &ptr[rb->buffer_size - rb->write_index],
81                                  length - (rb->buffer_size - rb->write_index));
82                          rb->write_index = length - (rb->buffer_size - rb->write_index);
83                  }
84          }
85
86          return RT_TRUE;
87  }
88
89  /* 从环形buffer中读出数据 */
90  static rt_bool_t rb_get(struct rb* rb, rt_uint8_t *ptr, rt_uint16_t length)
91  {
92          rt_size_t size;
93
94          /* 判断是否有足够的数据 */
95          if (rb->read_index > rb->write_index)
96                  size = rb->buffer_size - rb->read_index + rb->write_index;
97          else
98                  size = rb->write_index - rb->read_index;
99
100         /* 没有足够的数据 */
101         if (size < length) return RT_FALSE;
102
103         if (rb->read_index > rb->write_index)
104         {
105                 if (rb->buffer_size - rb->read_index > length)
106                 {
107                         /* read_index的数据足够多，直接复制 */
108                         memcpy(ptr, &rb->buffer_ptr[rb->read_index], length);
109                         rb->read_index += length;
110                 }
111                 else
112                 {
113                         /* read_index的数据不够，需要分段复制 */
114                         memcpy(ptr, &rb->buffer_ptr[rb->read_index],
115                                 rb->buffer_size - rb->read_index);
116                         memcpy(&ptr[rb->buffer_size - rb->read_index], &rb->buffer_ptr[0],
117                                 length - rb->buffer_size + rb->read_index);
```

```
118                             rb->read_index = length - rb->buffer_size + rb->read_index;
119                     }
120             }
121             else
122             {
123                     /*
124                      * read_index要比write_index小，总的数据量够（前面已经有总数据量的判
125                      * 断），直接复制出数据。
126                      */
127                     memcpy(ptr, &rb->buffer_ptr[rb->read_index], length);
128                     rb->read_index += length;
129             }
130
131             return RT_TRUE;
132     }
133
134     /* 生产者线程入口 */
135     static void thread_entry(void* parameter)
136     {
137             rt_bool_t result;
138             rt_uint8_t data_buffer[BUFFER_ITEM];
139
140             while (1)
141             {
142                     /* 持有信号量 */
143                     rt_sem_take(sem, RT_WAITING_FOREVER);
144                     /* 从环buffer中获得数据 */
145                     result = rb_get(&working_rb, &data_buffer[0], BUFFER_ITEM);
146                     /* 释放信号量 */
147                     rt_sem_release(sem);
148
149                     if (result == RT_TRUE)
150                     {
151                             /* 获取数据成功，打印数据 */
152                             rt_kprintf("%s\n", data_buffer);
153                     }
154
155                     /* 做一个5 OS Tick的休眠 */
156                     rt_thread_delay(5);
157             }
158     }
159
160     /* worker线程入口 */
161     static void worker_entry(void* parameter)
162     {
163             rt_bool_t result;
164             rt_uint32_t index, setchar;
165             rt_uint8_t  data_buffer[BUFFER_ITEM];
166
167             setchar = 0x21;
168             while (1)
169             {
170                     /* 构造数据 */
```

```
171                 for(index = 0; index < BUFFER_ITEM; index++)
172                 {
173                         data_buffer[index] = setchar;
174                         if (++setchar == 0x7f)
175                                 setchar = 0x21;
176                 }
177 
178                 /* 持有信号量 */
179                 rt_sem_take(sem, RT_WAITING_FOREVER);
180                 /* 把数据放到环形buffer中 */
181                 result = rb_put(&working_rb, &data_buffer[0], BUFFER_ITEM);
182                 /* 释放信号量 */
183                 rt_sem_release(sem);
184 
185                 /* 放入成功，做一个10 OS Tick的休眠 */
186                 rt_thread_delay(10);
187         }
188 }
189 
190 int semaphore_buffer_worker_init()
191 {
192         /* 初始化ring buffer */
193         rb_init(&working_rb, working_buffer, BUFFER_SIZE);
194 
195         /* 创建信号量 */
196         sem = rt_sem_create("sem", 1, RT_IPC_FLAG_FIFO);
197         if (sem == RT_NULL)
198         {
199                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
200                 return 0;
201         }
202 
203         /* 创建线程1 */
204         tid = rt_thread_create("thread",
205                 thread_entry, RT_NULL, /* 线程入口是thread_entry, 入口参数是RT_NULL */
206                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
207         if (tid != RT_NULL)
208                 rt_thread_startup(tid);
209         else
210                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
211 
212         /* 创建线程2 */
213         worker = rt_thread_create("worker",
214                 worker_entry, RT_NULL, /* 线程入口是worker_entry, 入口参数是RT_NULL */
215                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
216         if (worker != RT_NULL)
217                 rt_thread_startup(worker);
218         else
219                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
220 
221         return 0;
222 }
223 
```

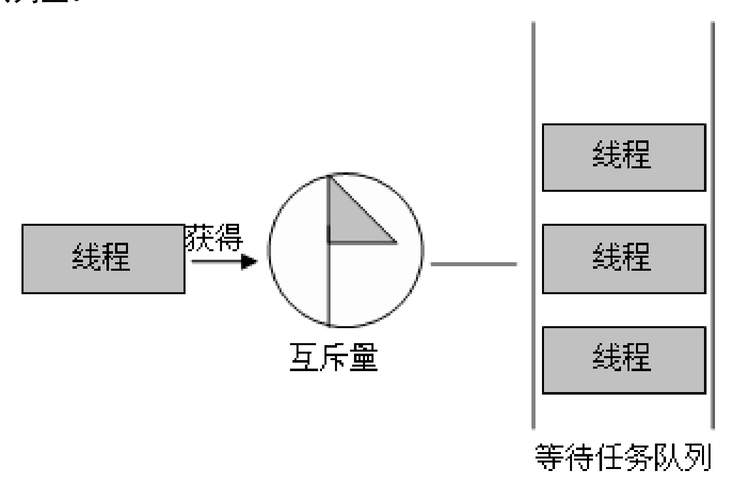
```
224 #ifdef RT_USING_TC
225 static void _tc_cleanup()
226 {
227         /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
228         rt_enter_critical();
229 
230         /* 删除信号量 */
231         if (sem != RT_NULL)
232                 rt_sem_delete(sem);
233 
234         /* 删除线程 */
235         if (tid != RT_NULL && tid->stat != RT_THREAD_CLOSE)
236                 rt_thread_delete(tid);
237         if (worker != RT_NULL && worker->stat != RT_THREAD_CLOSE)
238                 rt_thread_delete(worker);
239 
240         /* 调度器解锁 */
241         rt_exit_critical();
242 
243         /* 设置TestCase状态 */
244         tc_done(TC_STAT_PASSED);
245 }
246 
247 int _tc_semaphore_buffer_worker()
248 {
249         /* 设置TestCase清理回调函数 */
250         tc_cleanup(_tc_cleanup);
251         semaphore_buffer_worker_init();
252 
253         /* 返回TestCase运行的最长时间 */
254         return 100;
255 }
256 /* 输出函数命令到finsh shell中 */
257 FINSH_FUNCTION_EXPORT(_tc_semaphore_buffer_worker, a buffer worker with semaphore example);
258 #else
259 /* 用户应用入口 */
260 int rt_application_init()
261 {
262         semaphore_buffer_worker_init();
263 
264         return 0;
265 }
266 #endif
```

在这个例子中，semaphore是作为一种锁的形式存在的，当要访问临界资源：ring buffer时，通过持有semaphore的形式阻止其他线程进入（如果其他线程也打算进入，将在这里被挂起）。

7.4 互斥量

互斥量是管理临界资源的一种有效手段。因为互斥量是独占的，所以在一个时刻只允许一个线程占有互斥量，利用这个性质来实现共享资源的互斥量保护。互斥量工作示意图如下图所示，任何

时刻只允许一个线程获得互斥量对象，未能够获得互斥量对象的线程被挂起在该互斥量的等待线程队列上。



使用信号量会导致的一个潜在问题就是线程优先级翻转。所谓优先级翻转问题即当一个高优先级线程通过信号量机制访问共享资源时，该信号量已被一低优先级线程占有，而这个低优先级线程在运行过程中可能又被其它一些中等优先级的线程抢先，因此造成高优先级线程被许多具有较低优先级的线程阻塞，实时性难以得到保证。例如：有优先级为A、B和C的三个线程，优先级 $A > B > C$，线程A，B处于挂起状态，等待某一事件的发生，线程C正在运行，此时线程C开始使用某一共享资源S。在使用过程中，线程A等待的事件到来，线程A转为就绪态，因为它比线程C优先级高，所以立即执行。但是当线程A要使用共享资源S时，由于其正在被线程C使用，因此线程A被挂起切换到线程C运行。如果此时线程B等待的事件到来，则线程B转为就绪态。由于线程B的优先级比线程C高，因此线程B开始运行，直到其运行完毕，线程C才开始运行。只有当线程C释放共享资源S后，线程A才得以执行。在这种情况下，优先级发生了翻转，线程B先于线程A运行。这样便不能保证高优先级线程的响应时间。

在RT-Thread中实现的是优先级继承算法。优先级继承通过在线程A被阻塞期间提升线程C的优先级到线程A的优先级别从而解决优先级翻转引起的问题。这防止了C（间接地防止A）被B抢占。通俗地说，优先级继承协议使一个拥有资源的线程以等待该资源的线程中优先级最高的线程的优先级执行。当线程释放该资源时，它将返回到它正常的或标准的优先级。因此，继承优先级的线程避免了被任何中间优先级的线程抢占。

7.4.1 互斥量控制块

互斥量控制块的数据结构

```
struct rt_mutex
{
    struct rt_ipc_object parent;
    rt_base_t value;                /* 互斥量的数值          */
    struct rt_thread* owner;        /* 当前拥有互斥量的线程   */
    rt_uint8_t original_priority;   /* 线程原始优先级         */
    rt_base_t hold;                 /* 等待线程数量           */
};
```

rt_mutex对象从rt_ipc_object中派生，由IPC容器所管理。

7.4.2 互斥量相关接口

创建互斥量

创建一个互斥量时，内核首先创建一个互斥量控制块，然后完成对该控制块的初始化工作。创建互斥量使用以下接口：

```
rt_mutex_t rt_mutex_create (const char* name, rt_uint8_t flag)
```

使用该接口时，需要为互斥量指定一个名字，并指定互斥量标志，可以是基于优先级方式或基于FIFO方式：

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式       */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式     */
```

采用基于优先级flag创建的IPC对象，将在多个线程等待资源时，由优先级高的线程优先获得资源。而采用基于FIFO flag创建的IPC对象，在多个线程等待资源时，按照先来先得的顺序获得资源。

删除互斥量

系统不再使用互斥量时，通过删除互斥量以释放系统资源。删除互斥量使用以下接口：

```
rt_err_t rt_mutex_delete (rt_mutex_t mutex)
```

删除一个互斥量，所有等待此互斥量的线程都将被唤醒，等待线程获得返回值是-RT_ERROR。

初始化互斥量

静态互斥量对象的内存是在系统编译时由编译器分配的，一般放于数据段或ZI段中。在使用这类静态互斥量对象前，需要先对它进行初始化。初始化互斥量使用以下接口：

```
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8_t flag)
```

使用该接口时，需指定互斥量对象的句柄（即指向互斥量控制块的指针），指定该互斥量名称以及互斥量标志。互斥量标志可用值如上面的创建互斥量函数里的标志。

脱离互斥量

脱离互斥量将把互斥量对象从内核对象管理器中删除。脱离互斥量使用以下接口。

```
rt_err_t rt_mutex_detach (rt_mutex_t mutex)
```

使用该接口后，内核先唤醒所有挂在该互斥量上的线程（线程的返回值是-RT_ERROR），然后将该互斥量从内核对象管理器链表中删除。

获取互斥量

线程通过互斥量申请服务获取对互斥量的控制权。线程对互斥量的控制权是独占的，某一个时刻一个互斥量只能被一个线程控制。在RT-Thread中使用优先级继承算法来解决优先级翻转问题。成功获得该互斥量的线程的优先级将被提升到等待该互斥量资源的线程中优先级最高的线程的优先级，获取互斥量使用以下接口：

```
rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32_t time)
```

如果互斥量没有被其他线程控制，那么申请该互斥量的线程将成功获得。如果互斥量已经被当前线程线程控制，则该互斥量的引用计数加一。如果互斥量已经被其他线程控制，则当前线程该互斥量上挂起等待，直到其他线程释放它或者等待时间超过指定的超时时间。

释放互斥量

当线程完成互斥资源的访问后，应尽快释放它占据的互斥量，使得其他线程能及时获取该互斥量。释放互斥量使用以下接口：

```
rt_err_t rt_mutex_release(rt_mutex_t mutex)
```

使用该接口时，只有已经拥有互斥量控制权的线程才能释放它，每释放一次该互斥量，它的访问计数就减一。当该互斥量的访问计数为零时，它变为可用，等待在该信号量上的线程将被唤醒。如果线程的运行优先级被互斥量提升，那么当互斥量被释放后，线程恢复原先的优先级。

使用互斥量的例子如下

```
1  /*
2   * 程序清单：
3   */
4  #include <rtthread.h>
5  #include "tc_comm.h"
6
7  /* 指向线程控制块的指针 */
8  static rt_thread_t tid1 = RT_NULL;
9  static rt_thread_t tid2 = RT_NULL;
10 static rt_thread_t tid3 = RT_NULL;
11 static rt_mutex_t mutex = RT_NULL;
12
13 /* 线程1入口 */
14 static void thread1_entry(void* parameter)
15 {
16         /* 先让低优先级线程运行 */
17         rt_thread_delay(10);
18
19         /* 此时thread3持有mutex，并且thread2等待持有mutex */
20
21         /* 检查thread2与thread3的优先级情况 */
22         if (tid2->current_priority != tid3->current_priority)
23         {
24                 /* 优先级不相同，测试失败 */
25                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
26                 return;
```

```
27          }
28  }
29
30  /* 线程2入口 */
31  static void thread2_entry(void* parameter)
32  {
33          rt_err_t result;
34
35          /* 先让低优先级线程运行 */
36          rt_thread_delay(5);
37
38          while (1)
39          {
40                  /*
41                   * 试图持有互斥锁，此时thread3持有，应把thread3的优先级提升到thread2相同
42                   * 的优先级
43                   */
44                  result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
45
46                  if (result == RT_EOK)
47                  {
48                          /* 释放互斥锁 */
49                          rt_mutex_release(mutex);
50                  }
51          }
52  }
53
54  /* 线程3入口 */
55  static void thread3_entry(void* parameter)
56  {
57          rt_tick_t tick;
58          rt_err_t result;
59
60          while (1)
61          {
62                  result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
63                  result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
64                  if (result != RT_EOK)
65                  {
66                          tc_stat(TC_STAT_END | TC_STAT_FAILED);
67                  }
68
69                  /* 做一个长时间的循环，总共50个OS Tick */
70                  tick = rt_tick_get();
71                  while (rt_tick_get() - tick < 50) ;
72
73                  rt_mutex_release(mutex);
74                  rt_mutex_release(mutex);
75          }
76  }
77
78  int mutex_simple_init()
79  {
```

```
80	/* 创建互斥锁 */
81	mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
82	if (mutex == RT_NULL)
83	{
84		tc_stat(TC_STAT_END | TC_STAT_FAILED);
85		return 0;
86	}
87	
88	/* 创建线程1 */
89	tid1 = rt_thread_create("t1",
90		thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
91		THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
92	if (tid1 != RT_NULL)
93		rt_thread_startup(tid1);
94	else
95		tc_stat(TC_STAT_END | TC_STAT_FAILED);
96	
97	/* 创建线程2 */
98	tid2 = rt_thread_create("t2",
99		thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
100		THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
101	if (tid2 != RT_NULL)
102		rt_thread_startup(tid2);
103	else
104		tc_stat(TC_STAT_END | TC_STAT_FAILED);
105	
106	/* 创建线程3 */
107	tid3 = rt_thread_create("t3",
108		thread3_entry, RT_NULL, /* 线程入口是thread3_entry, 入口参数是RT_NULL */
109		THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
110	if (tid3 != RT_NULL)
111		rt_thread_startup(tid3);
112	else
113		tc_stat(TC_STAT_END | TC_STAT_FAILED);
114	
115	return 0;
116 }
117 
118 #ifdef RT_USING_TC
119 static void _tc_cleanup()
120 {
121	/* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
122	rt_enter_critical();
123	
124	/* 删除线程 */
125	if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
126		rt_thread_delete(tid1);
127	if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
128		rt_thread_delete(tid2);
129	if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)
130		rt_thread_delete(tid3);
131	
132	if (mutex != RT_NULL)
```

```
133         {
134                 rt_mutex_delete(mutex);
135         }
136 
137         /* 调度器解锁 */
138         rt_exit_critical();
139 
140         /* 设置TestCase状态 */
141         tc_done(TC_STAT_PASSED);
142 }
143 
144 int _tc_mutex_simple()
145 {
146         /* 设置TestCase清理回调函数 */
147         tc_cleanup(_tc_cleanup);
148         mutex_simple_init();
149 
150         /* 返回TestCase运行的最长时间 */
151         return 100;
152 }
153 /* 输出函数命令到finsh shell中 */
154 FINSH_FUNCTION_EXPORT(_tc_mutex_simple, sime mutex example);
155 #else
156 /* 用户应用入口 */
157 int rt_application_init()
158 {
159         mutex_simple_init();
160 
161         return 0;
162 }
163 #endif
```

7.5 事件

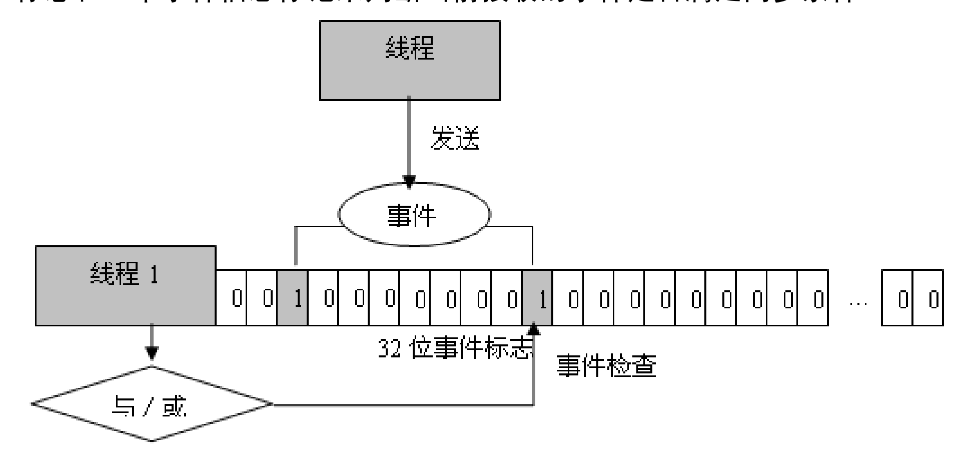
事件主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。即一个线程可等待多个事件的触发：可以是其中任一一个事件进行触发唤醒线程进行事件的处理操作；也可以是几个事件都到达后才触发唤醒线程进行后续的处理。同样，事件也可以是多个线程同步多个事件。这种多个事件的集合可以用一个32位无符号整型变量来表示，变量中的一位代表一个事件，线程通过"逻辑与"或"逻辑或"与一个或多个事件建立关联形成一个事件集。

事件的"逻辑或"也称为是独立型同步，指的是线程与任何事件之一发生同步；事件"逻辑与"也称为是关联型同步，指的是线程与若干事件都发生同步。

RT-Thread定义的事件有以下特点：

1. 事件只与线程相关，事件间相互独立：RT-Thread 定义的每个线程拥有32个事件标志，用一个32-bit无符号整型数记录，每一个bit代表一个事件。若干个事件构成一个事件集；
2. 事件仅用于同步，不提供数据传输功能；
3. 事件无排队性，即多次向线程发送同一事件(如果线程还未来得及读走)，其效果等同于只发送一次。

在RT-Thread实现中，每个线程还拥有一个事件信息标记，它有三个属性，分别是RT_EVENT_FLAG_AND（逻辑与），RT_EVENT_FLAG_OR（逻辑或）以及RT_EVENT_FLAG_CLEAR（清除标记）。当线程等待事件同步时，就可以通过32个事件标志和一个事件信息标记来判断当前接收的事件是否满足同步条件。



如上图所示，线程1的事件标志中第三位和第十位被置位，如果事件信息标记位设为逻辑与，则表示线程1只有在事件3和事件10都发生以后才会被触发唤醒，如果事件信息标记位设为逻辑或，则事件3或事件10中的任意一个发生都会触发唤醒线程1。如果信息标记同时设置了清除标记位，则发生的事件会导致线程1的相应事件标志位被重新置位为零。

7.5.1 事件控制块

```
struct rt_event
{
    struct rt_ipc_object parent;

    rt_uint32_t set;                /* 事件集合 */
};
```

rt_event对象从rt_ipc_object中派生，由IPC容器所管理。

7.5.2 事件相关接口

创建事件

当创建一个事件时，内核首先创建一个事件控制块，然后对该事件控制块进行基本的初始化，创建事件使用以下接口：

```
rt_event_t rt_event_create (const char* name, rt_uint8_t flag)
```

使用该接口时，需为事件指定名称以及事件标志。

，可以是基于优先级方式或基于FIFO方式：

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式      */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式    */
```

删除事件

系统不再使用事件对象时，通过删除事件对象控制块以释放系统资源。删除事件使用以下接口：

```
rt_err_t rt_event_delete (rt_event_t event)
```

删除一个事件，必须确保该事件不再被使用，同时将唤醒所有挂起在该事件上的线程（线程的返回值是-RT_ERROR）。

初始化事件

静态事件对象的内存是在系统编译时由编译器分配的，一般放于数据段或ZI段中。在使用静态事件对象前，需要先行对它进行初始化操作。初始化事件使用以下接口：

```
rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8_t flag)
```

使用该接口时，需指定静态事件对象的句柄（即指向事件控制块的指针），指定事件名称和事件标志。事件标志参数取值范围如上面的创建事件函数里的标志。

脱离事件

脱离信号量将把事件对象从内核对象管理器中删除。脱离事件使用以下接口。

```
rt_err_t rt_event_detach(rt_event_t event)
```

使用该接口后，内核首先唤醒所有挂在该事件上的线程（线程的返回值是-RT_ERROR），然后将该事件从内核对象管理器中删除。

接收事件

内核使用32位的无符号整型数来标识事件，它的每一位代表一个事件，因此一个事件对象可同时等待接收32个事件，内核可以通过指定选择参数“逻辑与”或“逻辑或”来选择如何激活线程，使用“逻辑与”参数表示只有当所有等待的事件都发生时激活线程，而使用“逻辑或”参数则表示只要有一个等待的事件发生就激活线程。接收事件使用以下接口：

```
rt_err_t rt_event_recv(rt_event_t event, rt_uint32_t set, rt_uint8_t option,
    rt_int32_t timeout, rt_uint32_t* recved)
```

用户线程首先根据选择参数和事件对象标志来判断它要接收的事件是否发生，如果已经发生，则根据线程要求选择是否重置事件对象的相应标志位，然后返回，如果没有发生，则把等待的事件标志位和选择参数填入线程本身的结构中，然后把线程挂起在此事件对象上，直到其等待的事件满足条件或等待时间超过指定超时时间。如果超时时间设置为零，则表示当线程要接受的事件没有满足其要求时不等待而直接返回。

发送事件

通过发送事件服务，可以发送一个或多个事件。发送事件使用以下接口：

```
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set)
```

使用该接口时，通过set参数指定的事件标志重新设定event对象的事件标志值，然后遍历等待在event事件上的线程链表，判断是否有线程的事件激活要求与当前event对象事件标志值匹配，如果有，则激活该线程。

使用事件的例子

```
/*
 * 程序清单：事件例程
 *
 * 这个程序会创建3个动态线程及初始化一个静态事件对象
 * 一个线程等于事件对象上以接收事件；
 * 一个线程定时发送事件 (事件3)
 * 一个线程定时发送事件 (事件5)
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

/* 事件控制块 */
static struct rt_event event;

/* 线程1入口函数 */
static void thread1_entry(void *param)
{
        rt_uint32_t e;

        while (1)
        {
                /* receive first event */
                if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
                        RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                        RT_WAITING_FOREVER, &e) == RT_EOK)
                {
                        rt_kprintf("thread1: AND recv event 0x%x\n", e);
                }

                rt_kprintf("thread1: delay 1s to prepare second event\n");
                rt_thread_delay(10);

                /* receive second event */
                if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
                        RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                        RT_WAITING_FOREVER, &e) == RT_EOK)
```

```
42                      {
43                              rt_kprintf("thread1: OR recv event 0x%x\n", e);
44                      }
45
46                      rt_thread_delay(5);
47              }
48      }
49
50      /* 线程2入口函数 */
51      static void thread2_entry(void *param)
52      {
53              while (1)
54              {
55                      rt_kprintf("thread2: send event1\n");
56                      rt_event_send(&event, (1 << 3));
57
58                      rt_thread_delay(10);
59              }
60      }
61
62      /* 线程3入口函数 */
63      static void thread3_entry(void *param)
64      {
65              while (1)
66              {
67                      rt_kprintf("thread3: send event2\n");
68                      rt_event_send(&event, (1 << 5));
69
70                      rt_thread_delay(20);
71              }
72      }
73
74      int event_simple_init()
75      {
76              /* 初始化事件对象 */
77              rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
78
79              /* 创建线程1 */
80              tid1 = rt_thread_create("t1",
81                      thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
82                      THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
83              if (tid1 != RT_NULL)
84                      rt_thread_startup(tid1);
85              else
86                      tc_stat(TC_STAT_END | TC_STAT_FAILED);
87
88              /* 创建线程2 */
89              tid2 = rt_thread_create("t2",
90                      thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
91                      THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
92              if (tid2 != RT_NULL)
93                      rt_thread_startup(tid2);
94              else
```

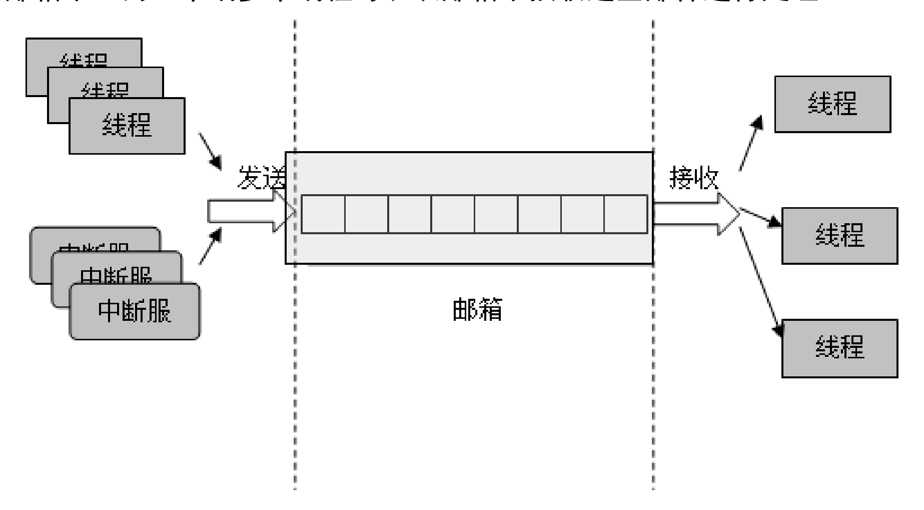
```
95                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
96
97          /* 创建线程3 */
98          tid3 = rt_thread_create("t3",
99                  thread3_entry, RT_NULL, /* 线程入口是thread3_entry, 入口参数是RT_NULL */
100                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
101         if (tid3 != RT_NULL)
102                 rt_thread_startup(tid3);
103         else
104                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
105
106         return 0;
107 }
108
109 #ifdef RT_USING_TC
110 static void _tc_cleanup()
111 {
112         /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
113         rt_enter_critical();
114
115         /* 删除线程 */
116         if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
117                 rt_thread_delete(tid1);
118         if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
119                 rt_thread_delete(tid2);
120         if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)
121                 rt_thread_delete(tid3);
122
123         /* 执行事件对象脱离 */
124         rt_event_detach(&event);
125
126         /* 调度器解锁 */
127         rt_exit_critical();
128
129         /* 设置TestCase状态 */
130         tc_done(TC_STAT_PASSED);
131 }
132
133 int _tc_event_simple()
134 {
135         /* 设置TestCase清理回调函数 */
136         tc_cleanup(_tc_cleanup);
137         event_simple_init();
138
139         /* 返回TestCase运行的最长时间 */
140         return 100;
141 }
142 /* 输出函数命令到finsh shell中 */
143 FINSH_FUNCTION_EXPORT(_tc_event_simple, a simple event example);
144 #else
145 /* 用户应用入口 */
146 int rt_application_init()
147 {
```

```
        event_simple_init();

        return 0;
}
#endif
```

7.6 邮箱

邮箱服务是实时操作系统中一种典型的任务间通信方法，通常开销比较低，效率较高，每一封邮件只能容纳固定的4字节内容（针对32位处理系统，刚好能够容纳一个指针）。典型的邮箱也称作交换消息，如下图所示，线程或中断服务例程把一则4字节长度（典型的是一个指针）的邮件发送到邮箱中。而一个或多个线程可以从邮箱中接收这些邮件进行处理。



RT-Thread采用的邮箱通信机制有点类型传统意义上的管道，用于线程间通讯。它是线程，中断服务，定时器向线程发送消息的有效手段。邮箱与线程对象等之间是相互独立的。线程，中断服务和定时器都可以向邮箱发送消息，但是只有线程能够接收消息（因为当邮箱为空时，线程将有可能被挂起）。RT-Thread的邮箱中共可存放固定条数的邮件，邮箱容量在创建邮箱时设定，每个邮件大小为4字节，正好是一个指针的大小。当需要在线程间传递比较大的消息时，可以传递指向一个缓冲区的指针。

当邮箱满时，线程等不再发送新邮件，返回-RT_EFULL。当邮箱空时，将可能挂起正在试图接收邮件的线程，使其等待，当邮箱中有新邮件时，再唤醒等待在邮箱上的线程，使其能够接收新邮件并继续后续的处理。

7.6.1 邮箱控制块

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;              /* 邮件池地址，用于存放邮件 */

    rt_size_t size;                     /* 邮件池大小              */
```

```
    rt_ubase_t entry;                   /* 邮箱中接收到的邮件数目   */
    rt_ubase_t in_offset, out_offset;   /* 邮件池的进/出偏移位置    */
};
```

rt_mailbox对象从rt_ipc_object中派生，由IPC容器所管理。

7.6.2 邮箱相关接口

创建邮箱

创建邮箱对象时先创建一个邮箱对象控制块，然后给邮箱分配一块内存空间用来存放邮件，这块内存大小等于邮件大小（4字节）与邮箱容量的乘积，然后接着初始化接收邮件和发送邮件在邮箱中的偏移量。创建邮箱的接口如下：

```
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8_t flag)
```

创建邮箱时需要给邮箱指定一个名称，作为邮箱的标识，并且指定邮箱的容量。flag参数可以选择基于优先级方式或基于FIFO方式：

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式      */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式    */
```

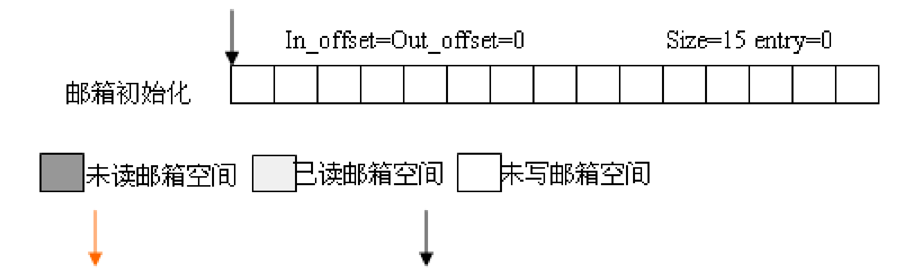
删除邮箱

当邮箱不再被使用时，应该删除它以释放相应的系统资源，一旦操作完成，邮箱将被永久性的删除。删除邮箱接口如下：

```
rt_err_t rt_mb_delete (rt_mailbox_t mb)
```

删除邮箱时，如果有线程被挂起在该邮箱对象上，则先唤醒挂起在该邮箱上的所有线程（线程获得返回值是-RT_ERROR），然后再释放邮箱使用的内存，最后删除邮箱对象。

初始化邮箱



初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态邮箱对象的初始化。其他与创建邮箱不同的是，此处静态邮箱对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给邮箱对象控制块，其余的初始化工作与创建邮箱时相同。接口如下：

```
rt_err_t rt_mb_init(rt_mailbox_t mb, const char* name, void* msgpool, rt_size_t size, rt_uint8_t flag)
```

初始化邮箱时，该接口需要获得用户已经申请获得的邮箱对象控制块以及缓冲区指针参数，以及线程指定的邮箱名和邮箱容量。如上图所示，邮箱初始化后接收邮件偏移量in_offset，out_offset均为零，邮箱容量size为15,邮箱中邮件数目entry为0。

> **Warning:** 这里的size参数指定的是邮箱的大小，即如果msgpool的字节数是N，那么邮箱大小最多为N/4。

脱离邮箱

脱离邮箱将把邮箱对象从内核对象管理器中删除。脱离邮箱使用以下接口。

```
rt_err_t rt_mb_detach(rt_mailbox_t mb)
```

使用该接口后，内核先唤醒所有挂在该邮箱上的线程（线程获得返回值是-RT_ERROR），然后将该邮箱对象从内核对象管理器中删除。

发送邮件

线程或者中断服务程序通过邮箱可以给其他线程发送邮件，发送的邮件可以是32位任意格式的数据，一个整型值或者指向一个缓冲区的指针。当邮箱中的邮件已经满时，发送邮件的线程或者中断程序会收到-RT_EFULL的返回值。发送邮件接口如下：

```
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32_t value)
```

发送者需指定接收邮箱具备，并且指定发送的邮件内容（value值）。

接收邮件

只有当接收者接收的邮箱中有邮件时，接收者才能立即取到邮件并返回RT_EOK的返回值，否则接收线程会根据超时时间设置或挂起在邮箱的等待线程队列上，或直接返回。接收邮件接口如下：

```
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32* value, rt_int32_t timeout)
```

接收邮件时，接收者需指定接收邮件的邮箱句柄，并指定接收到的邮件存放位置以及最多能够运行的超时时间。

> **Warning:** 只有线程能够接收邮箱中的邮件。

使用邮箱的例子

```
1 /*
2  * 程序清单：邮箱例程
3  *
4  * 这个程序会创建2个动态线程，一个静态的邮箱对象，其中一个线程往邮箱中发送邮件，
5  * 一个线程往邮箱中收取邮件。
6  */
7 #include <rtthread.h>
8 #include "tc_comm.h"
```

```
9
10  /* 指向线程控制块的指针 */
11  static rt_thread_t tid1 = RT_NULL;
12  static rt_thread_t tid2 = RT_NULL;
13
14  /* 邮箱控制块 */
15  static struct rt_mailbox mb;
16  /* 用于放邮件的内存池 */
17  static char mb_pool[128];
18
19  static char mb_str1[] = "I'm a mail!";
20  static char mb_str2[] = "this is another mail!";
21
22  /* 线程1入口 */
23  static void thread1_entry(void* parameter)
24  {
25          unsigned char* str;
26
27          while (1)
28          {
29                  rt_kprintf("thread1: try to recv a mail\n");
30
31                  /* 从邮箱中收取邮件 */
32                  if (rt_mb_recv(&mb, (rt_uint32_t*)&str, RT_WAITING_FOREVER) == RT_EOK)
33                  {
34                          rt_kprintf("thread1: get a mail from mailbox, the content:%s\n", str);
35
36                          /* 延时10个OS Tick */
37                          rt_thread_delay(10);
38                  }
39          }
40  }
41
42  /* 线程2入口 */
43  static void thread2_entry(void* parameter)
44  {
45          rt_uint8_t count;
46
47          count = 0;
48          while (1)
49          {
50                  count ++;
51                  if (count & 0x1)
52                  {
53                          /* 发送mb_str1地址到邮箱中 */
54                          rt_mb_send(&mb, (rt_uint32_t)&mb_str1[0]);
55                  }
56                  else
57                  {
58                          /* 发送mb_str2地址到邮箱中 */
59                          rt_mb_send(&mb, (rt_uint32_t)&mb_str2[0]);
60                  }
61
```

```
                /* 延时20个OS Tick */
                rt_thread_delay(20);
        }
}

int mbox_simple_init()
{
        /* 初始化一个mailbox */
        rt_mb_init(&mb,
                "mbt",                  /* 名称是mbt */
                &mb_pool[0],            /* 邮箱用到的内存池是mb_pool */
                size(mb_pool)/4,        /* 大小是mb_pool大小除以4，因为一封邮件的大小是4字节 */
                RT_IPC_FLAG_FIFO); /* 采用FIFO方式进行线程等待 */

        /* 创建线程1 */
        tid1 = rt_thread_create("t1",
                thread1_entry, RT_NULL, /* 线程入口是thread1_entry，入口参数是RT_NULL */
                THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
        if (tid1 != RT_NULL)
                rt_thread_startup(tid1);
        else
                tc_stat(TC_STAT_END | TC_STAT_FAILED);

        /* 创建线程2 */
        tid2 = rt_thread_create("t2",
                thread2_entry, RT_NULL, /* 线程入口是thread2_entry，入口参数是RT_NULL */
                THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
        if (tid2 != RT_NULL)
                rt_thread_startup(tid2);
        else
                tc_stat(TC_STAT_END | TC_STAT_FAILED);

        return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
        /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
        rt_enter_critical();

        /* 删除线程 */
        if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
                rt_thread_delete(tid1);
        if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
                rt_thread_delete(tid2);

        /* 执行邮箱对象脱离 */
        rt_mb_detach(&mb);

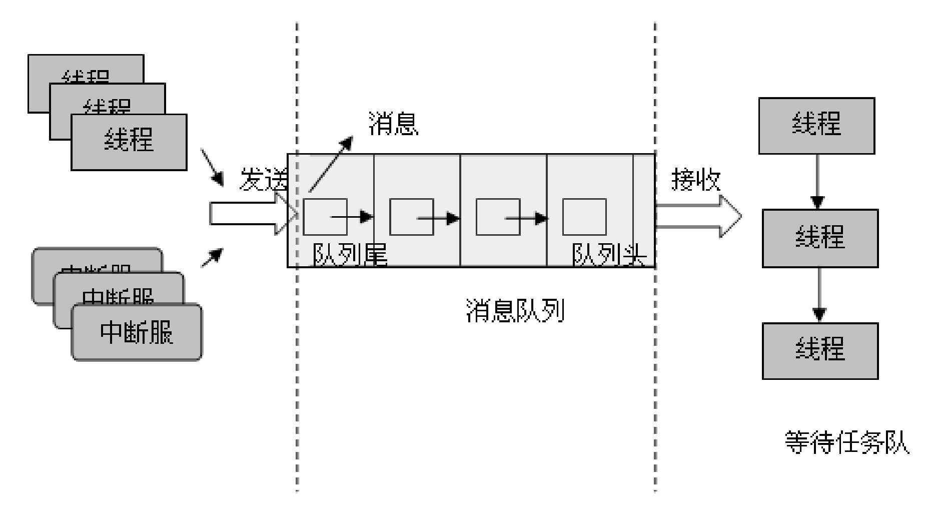
        /* 调度器解锁 */
        rt_exit_critical();

```

```
115         /* 设置TestCase状态 */
116         tc_done(TC_STAT_PASSED);
117 }
118 
119 int _tc_mbox_simple()
120 {
121         /* 设置TestCase清理回调函数 */
122         tc_cleanup(_tc_cleanup);
123         mbox_simple_init();
124 
125         /* 返回TestCase运行的最长时间 */
126         return 100;
127 }
128 /* 输出函数命令到finsh shell中 */
129 FINSH_FUNCTION_EXPORT(_tc_mbox_simple, a simple mailbox example);
130 #else
131 /* 用户应用入口 */
132 int rt_application_init()
133 {
134         mbox_simple_init();
135 
136         return 0;
137 }
138 #endif
```

7.7 消息队列

消息队列是另一种常用的线程间通讯方式，它能够接收来自线程的不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。而当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。



如上图所示，通过消息队列服务，线程或中断服务子程序可以将一条或多条消息放入消息队列。同样，一个或多个线程可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常先进入

消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。

RT-Thread的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：队列名，内存缓冲区，消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，每个消息框可以存放一条消息。消息队列中的第一个和最后一个消息框被分别称为队首和队尾，对应于消息队列控制块中的msg_queue_head和msg_queue_tail；有些消息框中可能是空的，它们通过msg_queue_free线程一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

7.7.1 消息队列控制块

```
struct rt_messagequeue
{
    struct rt_ipc_object parent;

    void* msg_pool;          /* 存放消息的消息池开始地址 */

    rt_size_t msg_size;      /* 每个消息的长度           */
    rt_size_t max_msgs;      /* 最大运行的消息数         */

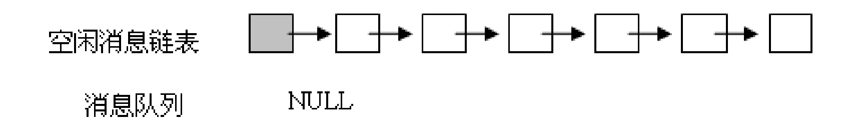
    void* msg_queue_head;    /* 消息链表头              */
    void* msg_queue_tail;    /* 消息链表尾              */
    void* msg_queue_free;    /* 空闲消息链表            */

    rt_ubase_t entry;        /* 队列中已经存放的消息数   */
};
```

rt_messagequeue对象从rt_ipc_object中派生，由IPC容器所管理。

7.7.2 消息队列相关接口

创建消息队列



创建消息队列时先创建一个消息队列对象控制块，然后给消息队列分配一块内存空间组织成空闲消息链表，这块内存大小等于（消息大小 + 消息头）与消息队列容量的乘积。然后再初始化消息队列，此时消息队列为空，如图所示。创建消息队列接口如下：

```
rt_mq_t rt_mq_create (const char* name, rt_size_t msg_size, rt_size_t max_msgs, rt_uint8_t flag)
```

创建消息队列时给消息队列指定一个名字，作为消息队列的标识，然后根据用户需求指定消息的大小以及消息队列的容量。如上图所示，消息队列被创建时所有消息都挂在空闲消息列表上，消息队列为空。flag标志，可以是基于优先级方式或基于FIFO方式：

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式      */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式    */
```

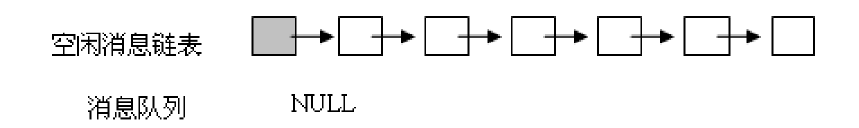
删除消息队列

当消息队列不再被使用时，应该删除它以释放系统资源，一旦操作完成，消息队列将被永久性的删除。删除消息队列接口如下：

```
rt_err_t rt_mq_delete (rt_mq_t mq)
```

删除消息队列时，如果有线程被挂起在该消息队列等待队列上，则先唤醒挂起在该消息等待队列上的所有线程（返回值是-RT_ERROR），然后再释放消息队列使用的内存，最后删除消息队列对象。

初始化消息队列



初始化静态消息队列对象跟创建消息队列对象类似，只是静态消息队列对象的内存是在系统编译时有编译器分配的，一般放于数据段或ZI段中。在使用这类静态消息队列对象前，需要先对它进行初始化。初始化消息队列对象接口如下：

```
rt_err_t rt_mq_init(rt_mq_t mq, const char* name, void *msgpool, rt_size_t msg_size,
        rt_size_t pool_size, rt_uint8_t flag)
```

初始化消息队列时，该接口需要获得消息队列对象的句柄（即指向消息队列对象控制块的指针），消息队列名，消息缓冲区指针，消息大小以及消息队列容量。如上图所示，消息队列初始化后所有消息都挂在空闲消息列表上，消息队列为空。此处的pool_size指的是消息缓冲区的大小，以字节数为单位。

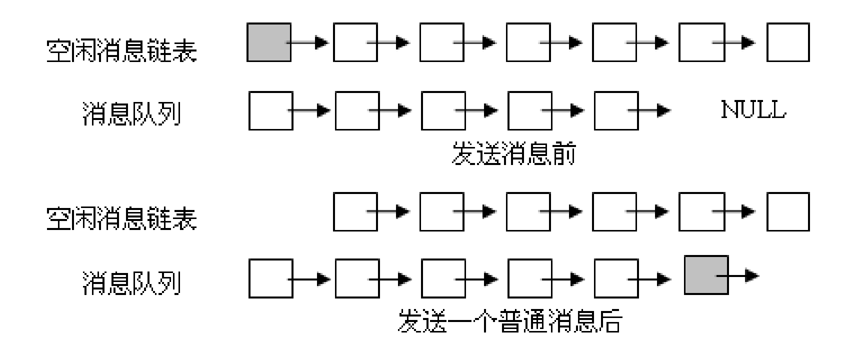
脱离消息队列

脱离消息队列将使消息队列对象被从内核对象管理器中删除。脱离消息队列使用以下接口。

```
rt_err_t rt_mq_detach(rt_mq_t mq)
```

使用该接口后，内核先唤醒所有挂在该消息等待队列对象上的线程（返回值是-RT_ERROR），然后将该消息队列对象从内核对象管理器中删除。

发送消息

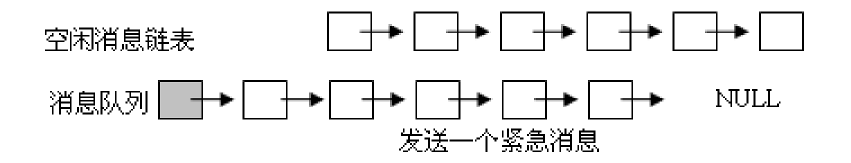


线程或者中断服务程序都可以给消息队列发送消息。当发送消息时，消息队列对象先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。发送者成功发送消息当且仅当空闲消息链表上有可用的空闲消息块；当自由消息链表上无可用消息块，说明消息队列中的消息已满，此时，发送消息的的线程或者中断程序会收到一个错误码（-RT_EFULL）。发送消息接口如下：

```
rt_err_t rt_mq_send (rt_mq_t mq, void* buffer, rt_size_t size)
```

发送消息时，发送者需指定发送到的消息队列对象句柄（即指向消息队列控制块的指针），并且指定发送的消息内容以及消息大小。上图所示，在发送一个普通消息之后，空闲消息链表上的队首消息被转移到了消息队列尾。

发送紧急消息

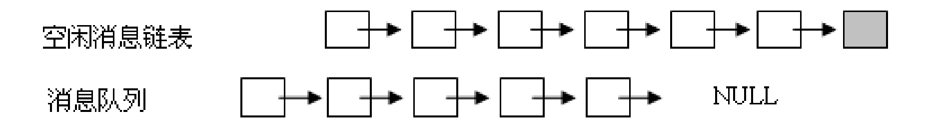


发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，从空闲消息链表上取下来的消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的接口如下：

```
rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size)
```

如上图所示，在发送一个紧急消息之后，空闲消息链表上的队首消息被转移到了消息队列首。

接收消息



当消息队列中有消息时，接收者才能接收消息，否则接收者会根据超时时间设置或挂起在消息队列的等待线程队列上，或直接返回。接收消息接口如下：

```
rt_err_t rt_mq_recv (rt_mq_t mq, void* buffer, rt_size_t size, rt_int32_t timeout)
```

接收消息时，接收者需指定存储消息的消息队列对象句柄，并且指定一个内存缓冲区，接收到的消息内容将被复制到该缓冲区上。此外，还需指定未能及时取到消息时的超时时间。如上图所示，接收一个消息后消息队列上的队首消息被转移到了空闲消息链表的尾部。

使用消息队列的例子

```
/*
 * 程序清单：消息队列例程
 *
 * 这个程序会创建3个动态线程，一个线程会从消息队列中收取消息；一个线程会定时给消
 * 息队列发送消息；一个线程会定时给消息队列发送紧急消息。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static char msg_pool[2048];

/* 线程1入口函数 */
static void thread1_entry(void* parameter)
{
        char buf[128];

        while (1)
        {
                rt_memset(&buf[0], 0, sizeof(buf));

                /* 从消息队列中接收消息 */
                if (rt_mq_recv(&mq, &buf[0], sizeof(buf), RT_WAITING_FOREVER) == RT_EOK)
                {
                        rt_kprintf("thread1: recv msg from message queue, the content:%s\n", buf);
                }

                /* 延迟10个OS Tick */
                rt_thread_delay(10);
        }
}

/* 线程2入口函数 */
static void thread2_entry(void* parameter)
{
        int i, result;
        char buf[] = "this is message No.x";
```

```
45
46          while (1)
47          {
48                  for (i = 0; i < 10; i++)
49                  {
50                          buf[sizeof(buf) - 2] = '0' + i;
51
52                          rt_kprintf("thread2: send message - %s\n", buf);
53                          /* 发送消息到消息队列中 */
54                          result = rt_mq_send(&mq, &buf[0], sizeof(buf));
55                          if ( result == -RT_EFULL)
56                          {
57                                  /* 消息队列满， 延迟1s时间 */
58                                  rt_kprintf("message queue full, delay 1s\n");
59                                  rt_thread_delay(100);
60                          }
61                  }
62
63                  /* 延时10个OS Tick */
64                  rt_thread_delay(10);
65          }
66  }
67
68  /* 线程3入口函数 */
69  static void thread3_entry(void* parameter)
70  {
71          char buf[] = "this is an urgent message!";
72
73          while (1)
74          {
75                  rt_kprintf("thread3: send an urgent message\n");
76
77                  /* 发送紧急消息到消息队列中 */
78                  rt_mq_urgent(&mq, &buf[0], sizeof(buf));
79
80                  /* 延时25个OS Tick */
81                  rt_thread_delay(25);
82          }
83  }
84
85  int messageq_simple_init()
86  {
87          /* 初始化消息队列 */
88          rt_mq_init(&mq, "mqt",
89                  &msg_pool[0], /* 内存池指向msg_pool */
90                  128 - sizeof(void*), /* 每个消息的大小是 128 - void* */
91                  sizeof(msg_pool), /* 内存池的大小是msg_pool的大小 */
92                  RT_IPC_FLAG_FIFO); /* 如果有多个线程等待，按照先来先得到的方法分配消息 */
93
94          /* 创建线程1 */
95          tid1 = rt_thread_create("t1",
96                  thread1_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
97                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
```

```
98          if (tid1 != RT_NULL)
99                  rt_thread_startup(tid1);
100         else
101                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
102
103         /* 创建线程2 */
104         tid2 = rt_thread_create("t2",
105                 thread2_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
106                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
107         if (tid2 != RT_NULL)
108                 rt_thread_startup(tid2);
109         else
110                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
111
112         /* 创建线程3 */
113         tid3 = rt_thread_create("t3",
114                 thread3_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
115                 THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
116         if (tid3 != RT_NULL)
117                 rt_thread_startup(tid3);
118         else
119                 tc_stat(TC_STAT_END | TC_STAT_FAILED);
120
121         return 0;
122 }
123
124 #ifdef RT_USING_TC
125 static void _tc_cleanup()
126 {
127         /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
128         rt_enter_critical();
129
130         /* 删除线程 */
131         if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
132                 rt_thread_delete(tid1);
133         if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
134                 rt_thread_delete(tid2);
135         if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)
136                 rt_thread_delete(tid3);
137
138         /* 执行消息队列对象脱离 */
139         rt_mq_detach(&mq);
140
141         /* 调度器解锁 */
142         rt_exit_critical();
143
144         /* 设置TestCase状态 */
145         tc_done(TC_STAT_PASSED);
146 }
147
148 int _tc_messageq_simple()
149 {
150         /* 设置TestCase清理回调函数 */
```

```
151         tc_cleanup(_tc_cleanup);
152         messageq_simple_init();
153 
154         /* 返回TestCase运行的最长时间 */
155         return 100;
156 }
157 /* 输出函数命令到finsh shell中 */
158 FINSH_FUNCTION_EXPORT(_tc_messageq_simple, a simple message queue example);
159 #else
160 /* 用户应用入口 */
161 int rt_application_init()
162 {
163         messageq_simple_init();
164 
165         return 0;
166 }
167 #endif
```

CHAPTER

EIGHT

内存管理

在计算系统中，变量、中间数据一般存放在系统存储空间中，当实际使用时才从存储空间调入到中央处理器内部进行运算。存储空间，按照存储方式来分类可以分为两种，内部存储空间和外部存储空间。内部存储空间通常访问速度比较快，能够随机访问（按照变量地址）。这一章主要讨论内部存储空间的管理。

实时系统中由于它对时间要求的严格性，其中的内存分配往往要比通用操作系统苛刻得多：

- 首先，分配内存的时间必须是确定性的。一般内存管理算法是搜索一个适当范围去寻找适合长度的空闲内存块。这个适当，造成了搜索时间的不确定性，这对于实时系统是不可接受的，因为实时系统必须要保证内存块的分配过程在可预测的确定时间内完成，否则实时任务在对外部事响应也将变得时间不可确定性，例如一个处理数据的例子：

当一个外部数据达到时（通过传感器或网络数据包），为了把它提交给上层的任务进行处理，它可能会先申请一块内存，把数据块的地址附加上，还可能有，数据长度以及一些其他信息附加在一起（放在一个结构体中），然后提交给上层任务。

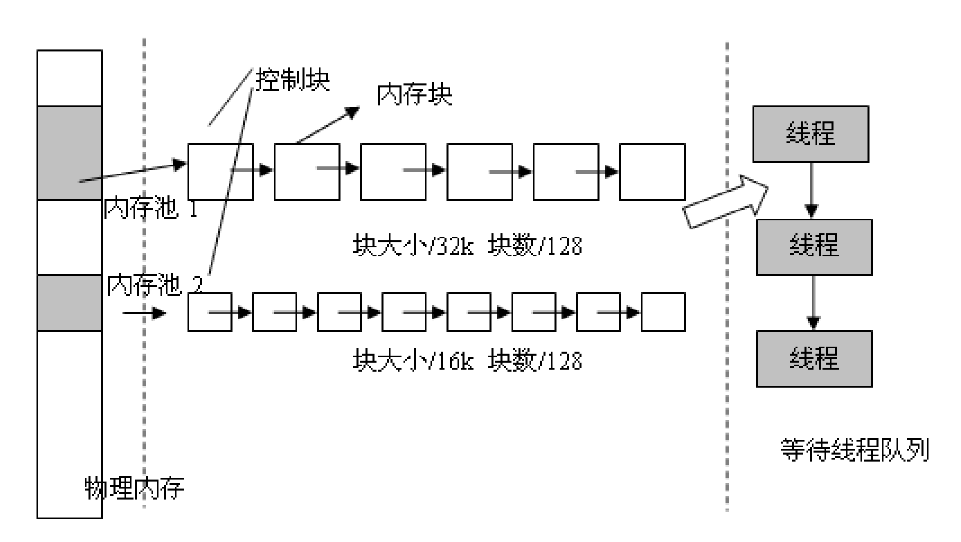
内存申请是当中的一个组成环节，如果因为使用的内存占用比较零乱，从而操作系统需要搜索一个不确定性长度的队列寻找适合的内存，那么申请时间将变得不可确定（可能搜索了1次，也可能搜索了若干次才能找到匹配的空间），进而对整个响应时间产生不可确定性。如果此时是一次导弹袭击，估计很可能会灰飞烟灭了！

- 其次，随着使用的内存分块被释放，整个内存区域会产生越来越多的碎片，从总体上来说，系统中还有足够的空闲内存，但因为它们非连续性，不能组成一块连续的完整内存块，从而造成程序不能申请到内存。对于通用系统而言，这种不恰当的内存分配算法可以通过重新启动系统来解决(每个月或者数个月进行一次)，但是这个对于可能需要数年工作于野外的嵌入式系统来说是不可接受的，他们通常需要连续不断地运行下去。
- 最后，嵌入式系统的资源环境也不是都相同，有些系统中资源比较紧张，只有数十KB的内存可供分配，而有些系统则存在数MB的内存。

RT-Thread操作系统在内存管理上，根据上层应用及系统资源的不同，有针对性的提供了数种内存分配管理算法：静态分区内存管理及动态内存管理。动态内存管理又更加可用内存多少划分为两种情况，一种是针对小内存块的分配管理，一种是针对大内存块的分配管理。

8.1 静态内存池管理

8.1.1 静态内存池工作模式



上图是内存池管理结构示意图。内存池（Memory Pool）是一种用于分配大量大小相同的小对象的技术。它可以极大加快内存分配/释放过程。

内存池在创建时向系统申请一大块内存，然后分成同样大小的多个小内存块，形成链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出头上一块，提供给申请者。如上图所示，物理内存中可以有多个大小不同的内存池，一个内存池由多个空闲内存块组成，内核用它们来进行内存管理。当一个内存池对象被创建时，内存池对象就被分配了内存池控制块：内存池名，内存缓冲区，内存块大小，块数以及一个等待线程队列。

内核负责给内存池分配内存池对象控制块，它同时也接收用户线程传入的参数，像内存块大小以及块数，由这些来确定内存池对象所需内存大小，当获得了这些信息后，内核就可以从内存堆或者线程私有内存空间中为内存池分配内存。内存池一旦初始化完成，内部的内存块大小将不能再做调整。

8.1.2 静态内存池控制块

```
struct rt_mempool
{
    struct rt_object parent;

    void* start_address;            /* 内存池数据区域开始地址                    */
    rt_size_t size;                 /* 内存池数据区域大小                        */

    rt_size_t block_size;           /* 内存块大小                          */
    rt_uint8_t* block_list;         /* 内存块列表                          */

    rt_size_t block_total_count;    /* 内存池数据区域中能够容纳的最大内存块数    */
    rt_size_t block_free_count;     /* 内存池中空闲的内存块数                    */

    rt_list_t suspend_thread;       /* 因为内存块不可用而挂起的线程列表     */
```

```
    rt_size_t suspend_thread_count; /* 因为内存块不可用而挂起的线程数            */
};
```

8.1.3 静态内存池接口

创建内存池

创建内存池操作将会创建一个内存池对象并且从堆上分配一个内存池。创建内存池是分配，释放内存块的基础，创建该内存池后，线程便可以从内存池中完成申请，释放操作，创建内存池使用如下接口，接口返回一个已创建的内存池对象。

```
rt_mp_t rt_mp_create(const char* name, rt_size_t block_count, rt_size_t block_size);
```

使用该接口可以创建与需求相匹配的内存块大小和数目的内存池，前提是在系统资源允许的情况下。创建内存池时，需要给内存池指定一个名称。根据需要，内核从系统中申请一个内存池对象，然后从内存堆中分配一块由块数目和块大小计算得来的内存大小，接着初始化内存池对象结构，并将申请成功的内存缓冲区组织成可用于分配的空闲块链表。

删除内存池

删除内存池将删除内存池对象并释放申请的内存。使用如下接口：

```
rt_err_t rt_mp_delete(rt_mp_t mp)
```

删除内存池时，必须首先唤醒等待在该内存池对象上的所有线程，然后再释放已从内存堆上分配的内存，然后删除内存池对象。

初始化内存池

初始化内存池跟创建内存池类似，只是初始化邮箱用于静态内存管理模式，内存池控制块来源于用户线程在系统中申请的静态对象。还与创建内存池不同的是，此处内存池对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给内存池对象控制块，其余的初始化工作与创建内存池相同。接口如下：

```
rt_err_t rt_mp_init(struct rt_mempool* mp, const char* name, void *start,
            rt_size_t size, rt_size_t block_size)
```

初始化内存池时，把需要进行初始化的内存池对象传递给内核，同时需要传递的还有内存池用到的内存空间，以及内存池管理的内存块数目和块大小，并且给内存池指定一个名称。这样，内核就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块链表。

脱离内存池

脱离内存池将使内存池对象被从内核对象管理器中删除。脱离内存池使用以下接口。

```
rt_err_t rt_mp_detach(struct rt_mempool* mp)
```

使用该接口后，内核先唤醒所有挂在该内存池对象上的线程，然后将内存池对象从内核对象管理器中删除。

分配内存块

从指定的内存池中分配一个内存块，使用如下接口：

```
void *rt_mp_alloc (rt_mp_t mp, rt_int32 time)
```

如果内存池中有可用的内存块，则从内存池的空闲块链表上取下一个内存块，减少空闲块数目并返回这个内存块，如果内存池中已经没有空闲内存块，则判断超时时间设置，若超时时间设置为零，则立刻返回空内存块，若等待大于零，则把当前线程挂起在该内存池对象上，直到内存池中有可用的自由内存块，或等待时间到达。

释放内存块

任何内存块使用完后都必须被释放，否则会造成内存泄露，释放内存块使用如下接口：

```
void rt_mp_free  (void *block)
```

使用以上接口时，首先通过需要被释放的内存块指针计算出该内存块所在的内存池对象，然后增加内存池对象的可用内存块数目，并把该被释放的内存块加入空闲内存块链表上。接着判断该内存池对象上是否有挂起的线程，如果有，则唤醒挂起线程链表上的首线程。

内存池的使用例子如下

```
#include <rtthread.h>

/* 两个线程用到的TCB和栈 */
struct rt_thread thread1;
struct rt_thread thread2;
char thread1_stack[512];
char thread2_stack[512];

/* 内存池数据存放区域 */
char mempool[4096];

/* 内存池TCB */
struct rt_mempool mp;

/* 测试用指针分配头 */
char *ptr[48];

/* 测试线程1入口 */
void thread1_entry(void* parameter)
{
    int i;
    char *block;

    while(1)
    {
```

```
        /* 分配48个内存块 */
        for (i = 0; i < 48; i++)
        {
            rt_kprintf("allocate No.%d\n", i);
            ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        }

        /* 再分配一个内存块 */
        block = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        rt_kprintf("allocate the block mem\n");
        /* 是否分配的内存块 */
        rt_mp_free(block);
        block = RT_NULL;
    }
}

/* 测试线程2入口 */
void thread2_entry(void *parameter)
{
    int i;

    while(1)
    {
        rt_kprintf("try to release block\n");

        /* 释放48个已经分配的内存块 */
        for (i = 0 ; i < 48; i ++)
        {
            /* 非空才释放 */
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("release block %d\n", i);
                rt_mp_free(ptr[i]);

                /* 释放完成，把指针清零 */
                ptr[i] = RT_NULL;
            }
        }
    }
}

int rt_application_init()
{
    int i;
    for (i = 0; i < 48; i ++) ptr[i] = RT_NULL;

    /* 初始化一个内存池对象，每个内存块的大小是80个字节 */
    rt_mp_init(&mp, "mp1", &mempool[0],
        sizeof(mempool), 80);

    /* 初始化两个测试线程对象 */
    rt_thread_init(&thread1,
        "thread1",
```

```
        thread1_entry, RT_NULL,
        &thread1_stack[0], sizeof(thread1_stack),
        20, 10);

    rt_thread_init(&thread2,
        "thread2",
        thread2_entry, RT_NULL,
        &thread2_stack[0], sizeof(thread2_stack),
        25, 7);
    rt_thread_startup(&thread1);
    rt_thread_startup(&thread2);

    return 0;
}
```

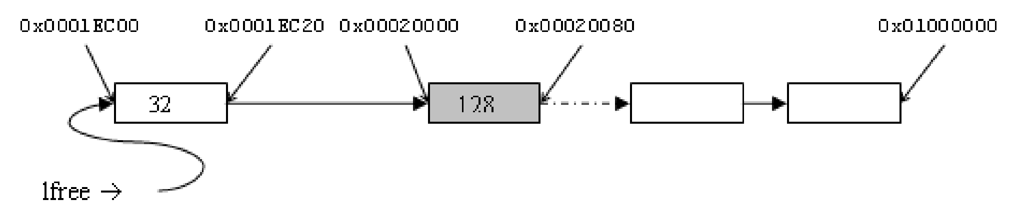
8.2 动态内存管理

动态内存管理是一个真实的堆内存管理模块，可以根据用户的需求（在当前资源满足的情况下）分配任意大小的内存块。RT-Thread系统中为了满足不同的需求，提供了两套动态内存管理算法，分别是小堆内存管理和SLAB内存管理。下堆内存管理模块主要针对系统资源比较少，一般小于2M内存空间的系统；而SLAB内存管理模块则主要是在系统资源比较丰富时，提供了一种近似的内存池管理算法。两种内存管理模块在系统运行时只能选择其中之一（或者完全不使用动态堆内存管理器），两种动态内存管理模块API形式完全相同。

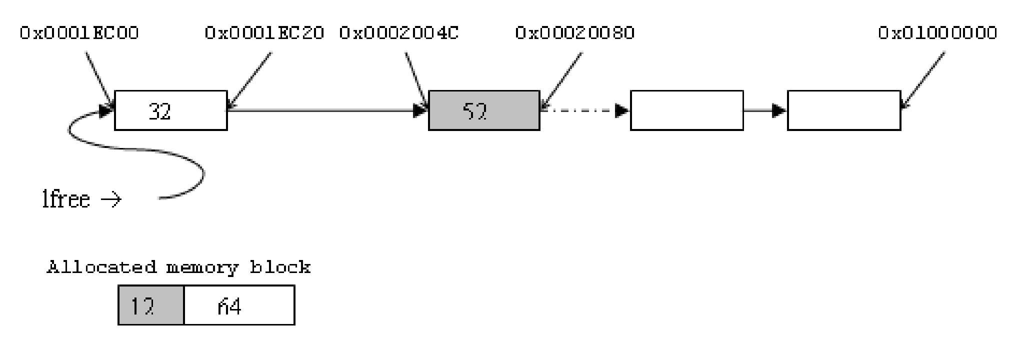
注：不要在中断服务例程中分配或释放动态内存块。

8.2.1 小内存管理模块

小内存管理算法是一个简单的内存分配算法，当有可用内存的时候，会从中分割出一块来作为分配的内存，而余下的则返回到动态内存堆中。如图4-5所示



当用户线程要分配一个64字节的内存块时，空闲链表指针lfree初始指向0x0001EC00内存块，但此内存块只有32字节并不能满足要求，它会继续寻找下一内存块，此内存块大小为128字节满足分配的要求。分配器将把此内存块进行拆分，余下的内存块（52字节）继续留在lfree链表中。如下图所示



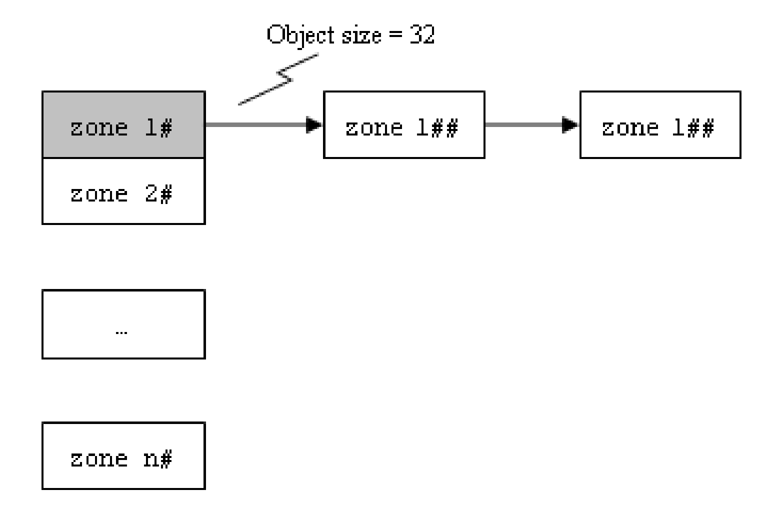
在分配的内存块前约12字节会存放内存分配器管理用的私有数据，用户线程不应访问修改它，这个头的大小会根据配置的对齐字节数稍微有些差别。

释放时则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

8.2.2 SLAB内存管理模块

RT-Thread 实现的SLAB分配器是在Matthew Dillon在DragonFly BSD中实现的SLAB分配器基础上针对嵌入式系统优化过的内存分配算法。原始的SLAB算法是Jeff Bonwick为 Solaris 操作系统首次引入的一种高效内核内存分配算法。

RT-Thread的SLAB分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。SLAB分配器会根据对象的类型（主要是大小）分成多个区（zone），也可以看成每类对象有一个内存池，如图所示：



一个zone的大小在32k ～ 128k字节之间，分配器会在堆初始化时根据堆的大小自动调整。系统中最多包括72种对象的zone，最大能够分配16k的内存空间，如果超出了16k那么直接从页分配器中分配。每个zone上分配的内存块大小是固定的，能够分配相同大小内存块的zone会链接在一个链表中，而72种对象的zone链表则放在一个数组（zone_arry）中统一管理。

动态内存分配器主要的两种操作：

- 内存分配：假设分配一个32字节的内存，SLAB内存分配器会先按照32字节的值，从zone_array链表表头数组中找到相应的zone链表。如果这个链表是空的，则向页分配器分配一个新的zone，然后从zone中返回第一个空闲内存块。如果链表非空，则这个zone链表

中的第一个zone节点必然有空闲块存在（否则它就不应该放在这个链表中），然后取相应的空闲块。如果分配完成后，导致一个zone中所有空闲内存块都使用完毕，那么分配器需要把这个zone节点从链表中删除。

- 内存释放：分配器需要找到内存块所在的zone节点，然后把内存块链接到zone的空闲内存块链表中。如果此时zone的空闲链表指示出zone的所有内存块都已经释放，即zone是完全空闲的zone。当中zone链表中，全空闲zone达到一定数目后，会把这个全空闲的zone释放到页面分配器中去。

8.2.3 动态内存接口

初始化系统堆空间

在使用堆内存时，必须要在系统初始化的时候进行堆内存的初始化，可以通过如下接口完成：

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

入口参数分别为堆内存的起始地址和结束地址。

分配内存块

从内存堆上分配用户线程指定大小的内存块，接口如下：

```
void* rt_malloc(rt_size_t nbytes);
```

用户线程需指定申请的内存空间大小，成功时返回分配的内存块地址，失败时返回RT_NULL。

重分配内存块

在已分配内存块的基础上重新分配内存块的大小（增加或缩小），可以通过如下接口完成：

```
void *rt_realloc(void *rmem, rt_size_t newsize);
```

在进行重新分配内存块时，原来的内存块数据保持不变（缩小的情况下，后面的数据被自动截断）。

分配多内存块

从内存堆中分配连续内存地址的多个内存块，可以通过如下接口完成：

```
void *rt_calloc(rt_size_t count, rt_size_t size);
```

返回的指针指向第一个内存块的地址，并且所有分配的内存块都被初始化成零。

释放内存块

用户线程使用完从内存分配器中申请的内存后，必须及时释放，否则会造成内存泄漏，释放接口如下：

```
void rt_free (void *ptr);
```

用户线程需传递待释放的内存块指针，如果是空指针直接返回。

设置分配钩子函数

在分配内存块过程中，用户可申请一个钩子函数，它会在内存分配完成后回调，接口如下：

```
void rt_malloc_sethook(void (*hook)(void *ptr, rt_size_t size));
```

回调时，会把分配到的内存块地址和大小做为入口参数传递进去。

设置内存释放钩子函数

在释放内存时，用户可设置一个钩子函数，它会在调用内存释放完成前进行回调，接口如下：

```
void rt_free_sethook(void (*hook)(void *ptr));
```

回调时，释放的内存块地址会做为入口参数传递进去（此时内存块并没有被释放）。

动态内存分配的例子

```
/* 线程TCB和栈 */
struct rt_thread thread1;
char thread1_stack[512];

/* 线程入口 */
void thread1_entry(void* parameter)
{
    int i;
    char *ptr[20]; /* 用于放置20个分配内存块的指针 */

    /* 对指针清零 */
    for (i = 0; i < 20; i ++) ptr[i] = RT_NULL;

    while(1)
    {
        for (i = 0; i < 20; i++)
        {
            /* 每次分配(1 << i)大小字节数的内存空间 */
            ptr[i] = rt_malloc(1 << i);

            /* 如果分配成功 */
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("get memory: 0x%x\n", ptr[i]);
```

```
                /* 释放内存块 */
                rt_free(ptr[i]);
                ptr[i] = RT_NULL;
            }
        }
    }
}

int rt_application_init()
{
    /* 初始化线程对象 */
    rt_thread_init(&thread1,
        "thread1",
        thread1_entry, RT_NULL,
        &thread1_stack[0], sizeof(thread1_stack),
        200, 100);

    rt_thread_startup(&thread1);

    return 0;
}
```

CHAPTER

NINE

异常与中断

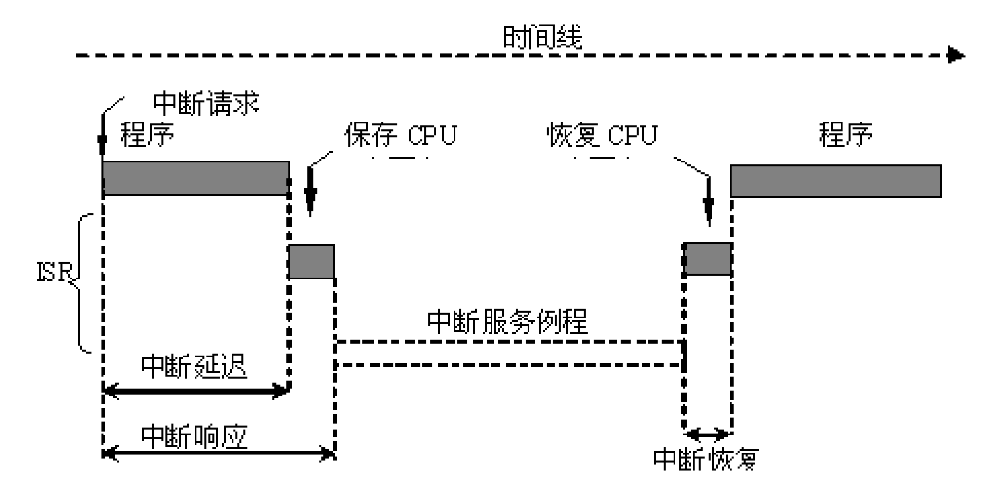
异常是导致处理器脱离正常运行转向执行特殊代码的任何事件，如果系统不及时处理，系统轻则出错，重着导致系统毁灭性的瘫痪。所以正确地处理异常避免错误的发生是提高软件的鲁棒性重要的一方面，对于嵌入式系统更加如此。

异常可以分成两类，同步异常和异步异常。同步异常主要是指由于内部事件产生的异常，例如除零错误。异步异常主要是指由于外部异常源产生的异常，例如按下设备某个按钮产生的事件。

中断，通常也叫做外部中断，中断属于异步异常。当中断源产生中断时，处理器也将同样陷入到一个固定位置去执行指令。

9.1 中断处理过程

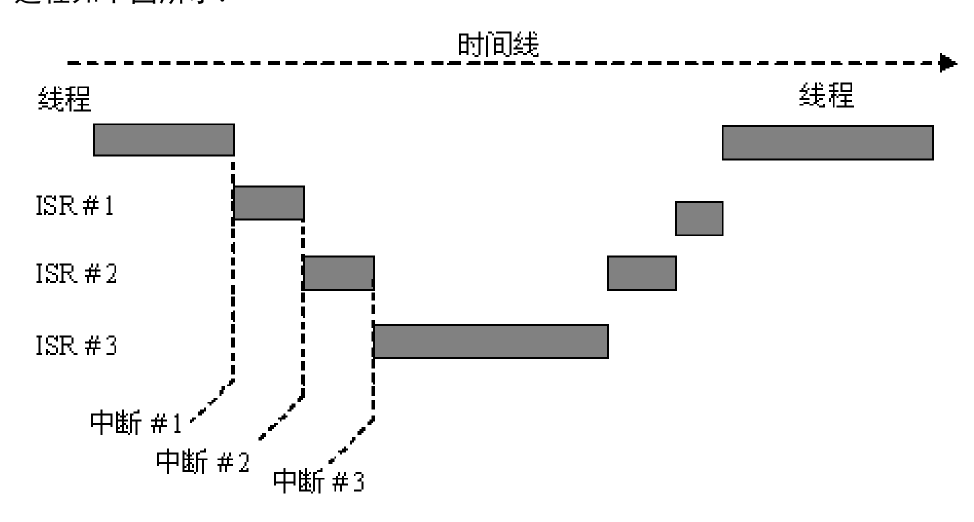
中断处理的一般过程如下图所示：



当中断产生时，处理机将按如下的顺序执行：

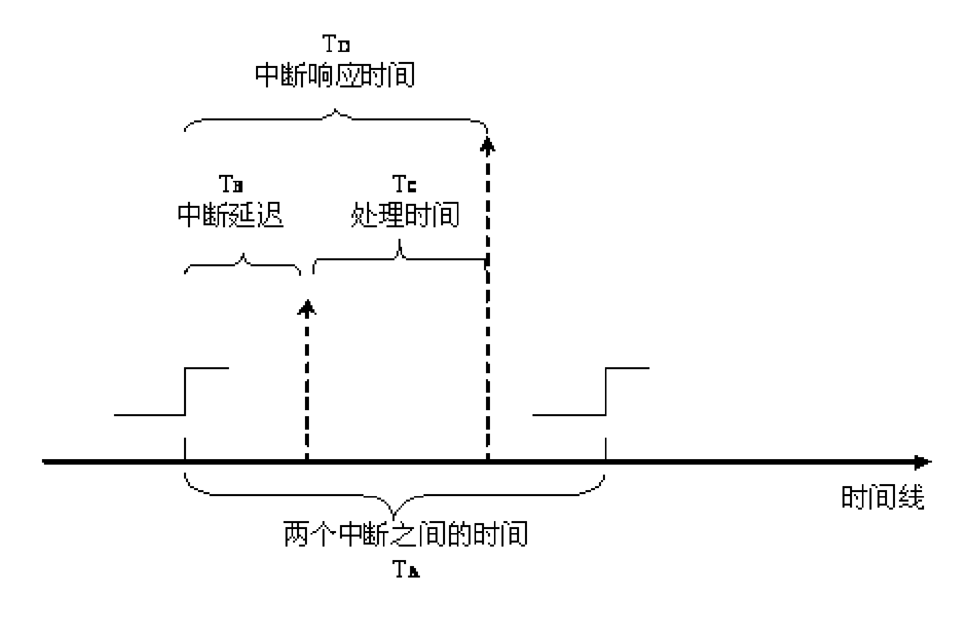
- 保存当前处理机状态信息
- 载入异常或中断处理函数到PC寄存器
- 把控制权转交给处理函数并开始执行
- 当处理函数执行完成时，恢复处理器状态信息
- 从异常或中断中返回到前一个程序执行点

中断使得CPU可以在事件发生时才予以处理，而不必让微处理器连续不断地查询是否有相应事件发生。通过两条特殊指令：关中断和开中断可以让处理器不响应或响应中断。在执行中断服务例程过程中，如果有更高优先级别的中断源触发中断，由于当前处于中断处理上下文环境中，根据不同的处理器构架可能有不同的处理方式：新的中断等待挂起直到当前中断处理离开或打断当前中断处理过程，让处理器相应这个更高优先级的中断源。后面这种情况，一般称之为中断嵌套。在硬实时环境中，前一种情况是不允许发生的，关闭中断响应的时间应尽量的短。在软件处理上，RT-Thread允许中断嵌套，即在一个中断服务例程期间，处理器可以响应另外一个更重要的中断，过程如下图所示：



当正在执行一个中断服务例程（中断1）时，有更高的中断触发，将保存当前中断服务例程的上下文环境，转向中断2的中断服务例程。当所有中断服务例程都运行完成时，才又恢复上下文环境转回到中断1的中断服务例程中接着执行。

即使如此，对于中断的处理仍然存在着（中断）时间响应的问题，先来看看中断处理过程中一个特定的时间量：



中断延迟TB定义为，从中断开始的时间到ISR程序开始执行的时间之间的时间段。而针对于处理

时间TC，这主要取决于ISR程序如何处理，而不同的设备其相应的服务程序的时间需求也不一样。

中断响应时间TD = TB + TC

RT-Thread提供独立的系统栈，即中断发生时，中断的前期处理程序会将用户的堆栈指针更换为系统事先留出的空间中，等中断退出时再恢复用户的堆栈指针。这样中断将不再占任务的堆栈空间，提高了内存空间的利用率，且随着任务的增加，这种技术的效果也越明显。

9.2 中断的底半处理

RT-Thread不对ISR所需要的处理时间做任何限制，但如同其它RTOS或非RTOS一样，用户需要保证所有的中断服务例程在尽可能短的时间内完成。这样在发生中断嵌套，或屏蔽了相应中断源的过程中，不会耽误了嵌套的其它中断处理过程，或自己中断源的下一次中断信号。

当一个中断信号发生时，ISR需要取得相应的硬件状态或者数据，如果ISR接下来要对状态或者数据进行简单处理，比如CPU时钟脉冲中断，ISR只需增加一个系统时钟tick，然后就结束ISR。这类中断往往所需的运行时间都比较短。对于另外一些中断，ISR在取得硬件状态或数据以后，还需要进行一系列更耗时的处理过程，通常需要将该中断分割为两部分，即上半部分（Top Half）和下半部分（Bottom Half）。在Top Half中，取得硬件状态和数据后，打开被屏蔽的中断，给相关的某个thread发送一条通知（可以是RT-Thread所提供的任意一种IPC方式），然后结束ISR。而接下来，相关的thread在接收到通知后，接着对状态或数据进行进一步的处理，这一过程称之为Bottom Half。

9.2.1 Bottom Half实现范例

在这一节中，为了详细描述Bottom Half在RT-Thread中的实现，我们以一个虚拟的网络设备接收网络数据包作为范例，并假设接收到数据报文后，对报文的分析、处理是一个相对耗时，比外部中断源信号重要性小许多，而且在不屏蔽中断源信号后也能处理的过程。

```
rt_sem_t demo_nw_isr;
void demo_nw_thread(void *param)
{
    /* 首先对设备进行必要的初始化工作 */
    device_init_setting();

    /* 装载中断服务例程 */
    rt_hw_interrupt_install(NW_IRQ_NUMBER, demo_nw_isr, RT_NULL);
    rt_hw_interrupt_umask(NW_IRQ_NUMBER);

    /*..其他的一些操作..*/

    /* 创建一个semaphore来响应Bottom Half的事件 */
    nw_bh_sem = rt_sem_create("bh_sem", 1, RT_IPC_FLAG_FIFO);

    while(1)
    {
        /* 最后，让demo_nw_thread等待在nw_bh_sem上 */
        rt_sem_take(nw_bh_sem, RT_WAITING_FOREVER);
```

```
        /* 接收到semaphore信号后，开始真正的Bottom Half处理过程 */
        nw_packet_parser (packet_buffer);
        nw_packet_process(packet_buffer);
    }
}

int rt_application_init()
{
    rt_thread_t thread;

    /* 创建处理线程 */
    thread = rt_thread_create("nwt",
    demo_nw_thread, RT_NULL,
    1024, 20, 5);

    if (thread != RT_NULL)
        rt_thread_startup(thread);
}
```

在上面代码中，创建了demo_nw_thread，并将thread阻塞在nw_bh_sem上，一旦semaphore被释放，将执行接下来的nw_packet_parser，开始Bottom Half的事件处理。接下来让我们来看一下demo_nw_isr中是如何处理Top Half，并开启Bottom Half的。

```
void demo_nw_isr(int vector)
{
    /* 当network设备接收到数据后，陷入中断异常，开始执行此ISR */
    /* 开始Top Half部分的处理，如读取硬件设备的状态以判断发生了何种中断 */
    nw_device_status_read();

    /*..一些其他操作等..*/

    /* 释放nw_bh_sem，发送信号给demo_nw_thread，准备开始Bottom Half */
    rt_sem_release(nw_bh_sem);

    /* 然后退出中断的Top Half部分，结束device的ISR */
}
```

由上面两个代码片段可以看出，通过一个IPC Object的等待和释放，来完成中断Bottom Half的起始和终结。由于将中断处理划分为Top和Bottom两个部分后，使得中断处理过程变为异步过程，这部分系统开销需要用户在使用RT-Thread时，必须认真考虑是否真正的处理时间大于给Bottom Half发送通知并处理的时间。

9.3 中断相关接口

RT-Thread为了把用户尽量的和系统底层异常、中断隔离开来，把中断和异常封装起来，提供给用户一个友好的接口。（注：这部分的API由BSP提供，在某些支持处理器支持分支中并不一定存在，例如ARM Cortex-M0/M3）

9.3.1 装载中断服务例程

通过调用rt_hw_interrupt_install，把用户的中断服务例程(new_handler)和指定的中断号关联起来，当这个中断源产生中断时，系统将自动调用装载的中断服务例程。如果old_handler不为空，则把之前关联的这个中断服务例程卸载掉。接口如下：

```
void rt_hw_interrupt_install(int vector, rt_isr_handler_t new_handler, rt_isr_handler_t *old_handler)
```

Note: 这个API并不出现在Cortex-M0/M3的移植分支中。

9.3.2 屏蔽中断源

通常，在ISR准备处理某个中断信号之前，需要屏蔽该中断源，以保证在接下来的处理过程中硬件状态或者数据不会遭到干扰。接口如下：

```
void rt_hw_interrupt_mask(int vector)
```

Note: 这个API并不出现在Cortex-M0/M3的移植分支中。

9.3.3 打开被屏蔽的中断源

在ISR处理完状态或数据以后，需要及时的打开之前被屏蔽的中断源，使得尽可能的不丢失硬件中断信号。接口如下：

```
void rt_hw_interrupt_umask(int vector)
```

Note: 这个API并不出现在Cortex-M0/M3的移植分支中。

9.3.4 关闭中断

当需要关闭中断以屏蔽整个系统的事件处理时，调用如下接口：

```
rt_base_t rt_hw_interrupt_disable()
```

当系统关闭了中断时，就意味着当前任务/代码不会被其他事件所打断（因为整个系统已经对外部事件不再响应），也就是当前任务不会被抢占，除非这个任务主动退出处理机。

9.3.5 打开中断

打开中断往往是和关闭中断成对使用的，用于恢复关闭中断前的状态。接口如下：

```
void rt_hw_interrupt_enable(rt_base_t level)
```

调用这个接口将恢复调用rt_hw_interrupt_disable前的中断状态，level是上一次关闭中断时返回的值。

注：调用这个接口并不代表着肯定会打开中断，而是恢复关闭中断前的状态，如果调用rt_hw_interrupt_disable（）前是关中断状态，那么调用此函数后依然是关中断状态。

CHAPTER

TEN

定时器与系统时钟

10.1 定时器管理

定时器，是指在指定的时刻开始，经过一指定的时间后出发一个事件。定时器有硬件定时器和软件定时器之分：

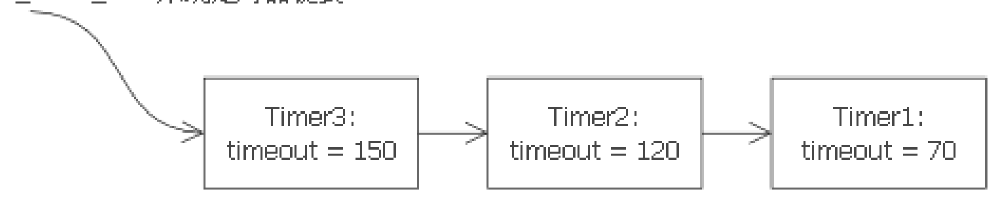
- 硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。
- 软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。软件定时器的精度取决于它使用的硬件定时器精度，例如硬件定时器精度是1秒，那么它能够提供1秒，5秒，8秒等以1秒整数倍的定时器，而不能够提供1.5秒的定时器。

在RT-Thread实时操作系统中，软件定时器模块以tick为时间单位，tick的时间长度为两次硬件定时器中断的时间间隔，这个时间可以根据不同的系统MIPS和实时性需求设置不同的值，tick值设置越小，实时精度越高，但是系统开销也越大。

RT-Threa的软定时器提供两类定时器机制：第一类是单次触发定时器，这类定时器只会触发一次定时器事件，然后定时器自动停止。第二类则是周期触发定时器，这类定时器会周期性的触发定时器事件。

下面以实际例子来说明RT-Thread软件定时器的基本工作原理，在RT-Thread定时器模块维护两个重要的全局变量，一个是当前系统的时间rt_tick（当硬件定时器中断来临时，它将加1），另一个是定时器链表rt_timer_list，系统中新创建的定时期都会被以排序的方式插入到rt_timer_list链表中。

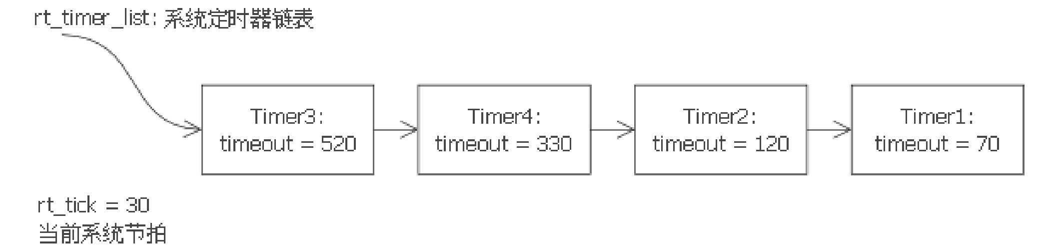




如上图所示，系统当前tick值为20，在当前系统中已经创建并启动了三个定时器，分别为定时时间为50个tick的Timer1、100个tick的Timer2和500个tick的Timer3，这三个定时器分别被加上系统当前时间rt_tick = 20后从小到大排序插入到rt_timer_list链表中，形成如上图所示的定时器链表结构。

而rt_tick随着硬件定时器的触发一直在增长，50个tick以后，rt_tick从20增长到70，与Timer1的timeout值相等，这时会触发Timer1定时期相关连的超时函数，同时将Timer1从rt_timer_list链表上删除。同理，100个tick和500个tick过去后，Timer2和Timer3定时器相关联的超时函数会被触发，接着将Time2和Timer3定时器从rt_timer_list链表中删除。

如果系统当前定时器状态如上图中，10个tick以后，rt_tick = 30，此时有任务新创建一个tick值为300的Timer4定时器，则Timer4定时器的timeout = rt_tick + 300 =330，然后被插入到Timer2和Timer3定时器中间，形成如下图所示链表结构。



10.2 定时器管理控制块

```
struct rt_timer
{
    struct rt_object parent;
    rt_list_t list;                              /* 用于链接定时器的链表      */
    void (*timeout_func)(void* parameter);       /* 定时器超时调用的函数      */
    void *parameter;                             /* 超时函数用到的入口参数    */
    rt_tick_t init_tick;                         /* 定时器初始超时节拍数      */
    rt_tick_t timeout_tick;                      /* 定时器实际超时时的节拍数  */
};
```

10.3 定时器管理接口

10.3.1 定时器管理系统初始化

初始化定时器管理系统，可以通过如下接口完成：

```
void rt_system_timer_init()
```

10.3.2 创建定时器

当动态创建一个定时器时，内核首先创建一个定时器控制块，然后对该控制块进行基本的初始化，创建定时器使用以下接口：

```
rt_timer_t rt_timer_create(const char* name, void (*timeout)(void* parameter),
        void* parameter, rt_tick_t time, rt_uint8_t flag)
```

使用该接口时，需要为定时器指定名称，提供定时器回调函数及参数，定时时间，并指定是单次定时还是周期定时。创建定时器的例子如下代码所示。

```c
/*
 * 程序清单：动态定时器例程
 *
 * 这个例程会创建两个动态定时器对象，一个是单次定时，一个是周期性的定时
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_timer_t timer2;

/* 定时器1超时函数 */
static void timeout1(void* parameter)
{
        rt_kprintf("periodic timer is timeout\n");
}

/* 定时器2超时函数 */
static void timeout2(void* parameter)
{
        rt_kprintf("one shot timer is timeout\n");
}

void timer_create_init()
{
        /* 创建定时器1 */
        timer1 = rt_timer_create("timer1",  /* 定时器名字是 timer1 */
                timeout1, /* 超时时回调的处理函数 */
                RT_NULL, /* 超时函数的入口参数 */
                10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
                RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
        /* 启动定时器 */
        if (timer1 != RT_NULL)
                rt_timer_start(timer1);
        else
                tc_stat(TC_STAT_END | TC_STAT_FAILED);

        /* 创建定时器2 */
        timer2 = rt_timer_create("timer2",   /* 定时器名字是 timer2 */
                timeout2, /* 超时时回调的处理函数 */
                RT_NULL, /* 超时函数的入口参数 */
                30, /* 定时长度为30个OS Tick */
                RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

        /* 启动定时器 */
        if (timer2 != RT_NULL)
                rt_timer_start(timer2);
        else
                tc_stat(TC_STAT_END | TC_STAT_FAILED);
}

```

```
53 #ifdef RT_USING_TC
54 static void _tc_cleanup()
55 {
56         /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
57         rt_enter_critical();
58 
59         /* 删除定时器对象 */
60         rt_timer_delete(timer1);
61         rt_timer_delete(timer2);
62 
63         /* 调度器解锁 */
64         rt_exit_critical();
65 
66         /* 设置TestCase状态 */
67         tc_done(TC_STAT_PASSED);
68 }
69 
70 int _tc_timer_create()
71 {
72         /* 设置TestCase清理回调函数 */
73         tc_cleanup(_tc_cleanup);
74 
75         /* 执行定时器例程 */
76         timer_create_init();
77 
78         /* 返回TestCase运行的最长时间 */
79         return 100;
80 }
81 /* 输出函数命令到finsh shell中 */
82 FINSH_FUNCTION_EXPORT(_tc_timer_create, a dynamic timer example);
83 #else
84 /* 用户应用入口 */
85 int rt_application_init()
86 {
87         timer_create_init();
88 
89         return 0;
90 }
91 #endif
```

10.3.3 删除定时器

系统不再使用特定定时器时，通过删除该定时器以释放系统资源。删除定时器使用以下接口：

```
rt_err_t rt_timer_delete(rt_timer_t timer)
```

删除定时器的例子请参考创建定时器部分代码。

10.3.4 初始化定时器

当选择静态创建定时器时，可利用rt_timer_init接口来初始化该定时器，接口如下：

```
void rt_timer_init(rt_timer_t timer, const char* name, void (*timeout)(void* parameter),
        void* parameter, rt_tick_t time, rt_uint8_t flag)
```

使用该接口时，需指定定时器对象，定时器名称，提供定时器回调函数及参数，定时时间，并指定是单次定时还是周期定时。初始化定时器的例子如下代码所示。

```
/*
 * 程序清单：定时器例程
 *
 * 这个程序会初始化2个静态定时器，一个是单次定时，一个是周期性的定时
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 定时器的控制块 */
static struct rt_timer timer1;
static struct rt_timer timer2;

/* 定时器1超时函数 */
static void timeout1(void* parameter)
{
        rt_kprintf("periodic timer is timeout\n");
}

/* 定时器2超时函数 */
static void timeout2(void* parameter)
{
        rt_kprintf("one shot timer is timeout\n");
}

void timer_static_init()
{
        /* 初始化定时器 */
        rt_timer_init(&timer1, "timer1",  /* 定时器名字是 timer1 */
                timeout1, /* 超时时回调的处理函数 */
                RT_NULL, /* 超时函数的入口参数 */
                10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
                RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
        rt_timer_init(&timer2, "timer2",   /* 定时器名字是 timer2 */
                timeout2, /* 超时时回调的处理函数 */
                RT_NULL, /* 超时函数的入口参数 */
                30, /* 定时长度为30个OS Tick */
                RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

        /* 启动定时器 */
        rt_timer_start(&timer1);
        rt_timer_start(&timer2);
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
```

```
47          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
48          rt_enter_critical();
49
50          /* 执行定时器脱离 */
51          rt_timer_detach(&timer1);
52          rt_timer_detach(&timer2);
53
54          /* 调度器解锁 */
55          rt_exit_critical();
56
57          /* 设置TestCase状态 */
58          tc_done(TC_STAT_PASSED);
59  }
60
61  int _tc_timer_static()
62  {
63          /* 设置TestCase清理回调函数 */
64          tc_cleanup(_tc_cleanup);
65
66          /* 执行定时器例程 */
67          timer_static_init();
68
69          /* 返回TestCase运行的最长时间 */
70          return 100;
71  }
72  /* 输出函数命令到finsh shell中 */
73  FINSH_FUNCTION_EXPORT(_tc_timer_static, a static timer example);
74  #else
75  /* 用户应用入口 */
76  int rt_application_init()
77  {
78          timer_static_init();
79
80          return 0;
81  }
82  #endif
```

10.3.5 脱离定时器

脱离定时器使定时器对象被从系统容器的链表中脱离出来，但定时器对象所占有的内存不会被释放，脱离信号量使用以下接口。

```
rt_err_t rt_timer_detach(rt_timer_t timer)
```

脱离定时器的例子可参考定时器初始化代码中的脱离部分。

10.3.6 启动定时器

当定时器被创建或者初始化以后，不会被立即启动，必须在调用启动定时器接口后，才开始工作，启动定时器接口如下：

```
rt_err_t rt_timer_start(rt_timer_t timer)
```

启动定时器的例子请参考定时器初始化例程代码。

10.3.7 停止定时器

启动定时器以后，若想使它停止，可以使用该接口：

```
rt_err_t rt_timer_stop(rt_timer_t timer)
```

停止定时器的例子如下代码所示。

```
/*
 * 程序清单：动态定时器例程
 *
 * 这个例程会创建1个动态周期型定时器对象
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_uint8_t count;

/* 定时器超时函数 */
static void timeout1(void* parameter)
{
	rt_kprintf("periodic timer is timeout\n");

	count ++;
	/* 停止定时器自身 */
	if (count >= 8)
	{
		/* 停止定时器 */
		rt_timer_stop(timer1);
		count = 0;
	}
}

void timer_stop_self_init()
{
	/* 创建定时器1 */
	timer1 = rt_timer_create("timer1",  /* 定时器名字是 timer1 */
		timeout1, /* 超时时回调的处理函数 */
		RT_NULL, /* 超时函数的入口参数 */
		10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
		RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
	/* 启动定时器 */
	if (timer1 != RT_NULL)
		rt_timer_start(timer1);
	else
```

```
40                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
41  }
42
43  #ifdef RT_USING_TC
44  static void _tc_cleanup()
45  {
46          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
47          rt_enter_critical();
48
49          /* 删除定时器对象 */
50          rt_timer_delete(timer1);
51          timer1 = RT_NULL;
52
53          /* 调度器解锁 */
54          rt_exit_critical();
55
56          /* 设置TestCase状态 */
57          tc_done(TC_STAT_PASSED);
58  }
59
60  int _tc_timer_stop_self()
61  {
62          /* 设置TestCase清理回调函数 */
63          tc_cleanup(_tc_cleanup);
64
65          /* 执行定时器例程 */
66          count = 0;
67          timer_stop_self_init();
68
69          /* 返回TestCase运行的最长时间 */
70          return 100;
71  }
72  /* 输出函数命令到finsh shell中 */
73  FINSH_FUNCTION_EXPORT(_tc_timer_stop_self, a dynamic timer example);
74  #else
75  /* 用户应用入口 */
76  int rt_application_init()
77  {
78          timer_stop_self_init();
79
80          return 0;
81  }
82  #endif
```

10.3.8 控制定时器

控制定时器接口可以用来查看或改变定时器的设置，它提供四个命令接口，分别是设置定时时间，查看定时时间，设置单次触发，设置周期触发。命令如下：

```
#define RT_TIMER_CTRL_SET_TIME          0x0     /* 设置定时器超时时间        */
#define RT_TIMER_CTRL_GET_TIME          0x1     /* 获得定时器超时时间        */
```

```
#define RT_TIMER_CTRL_SET_ONESHOT       0x2     /* 设置定时器为单一超时型   */
#define RT_TIMER_CTRL_SET_PERIODIC      0x3     /* 设置定时器为周期型定时器 */
```

控制定时器接口如下：

```
rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg)
```

使用该接口时，需指定定时器对象，控制命令及相应参数。控制定时器的例子如下代码所示。

```
/*
 * 程序清单：动态定时器例程
 *
 * 这个例程会创建1个动态周期型定时器对象，然后控制它进行定时时间长度的更改。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_uint8_t count;

/* 定时器超时函数 */
static void timeout1(void* parameter)
{
        rt_kprintf("periodic timer is timeout\n");

        count ++;
        /* 停止定时器自身 */
        if (count >= 8)
        {
                /* 控制定时器然后更改超时时间长度 */
                rt_timer_control(timer1, RT_TIMER_CTRL_SET_TIME, (void*)50);
                count = 0;
        }
}

void timer_control_init()
{
        /* 创建定时器1 */
        timer1 = rt_timer_create("timer1",  /* 定时器名字是 timer1 */
                timeout1, /* 超时时回调的处理函数 */
                RT_NULL, /* 超时函数的入口参数 */
                10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
                RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
        /* 启动定时器 */
        if (timer1 != RT_NULL)
                rt_timer_start(timer1);
        else
                tc_stat(TC_STAT_END | TC_STAT_FAILED);
}

#ifdef RT_USING_TC
static void _tc_cleanup()
```

```
45  {
46          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
47          rt_enter_critical();
48
49          /* 删除定时器对象 */
50          rt_timer_delete(timer1);
51          timer1 = RT_NULL;
52
53          /* 调度器解锁 */
54          rt_exit_critical();
55
56          /* 设置TestCase状态 */
57          tc_done(TC_STAT_PASSED);
58  }
59
60  int _tc_timer_control()
61  {
62          /* 设置TestCase清理回调函数 */
63          tc_cleanup(_tc_cleanup);
64
65          /* 执行定时器例程 */
66          count = 0;
67          timer_control_init();
68
69          /* 返回TestCase运行的最长时间 */
70          return 100;
71  }
72  /* 输出函数命令到finsh shell中 */
73  FINSH_FUNCTION_EXPORT(_tc_timer_control, a timer control example);
74  #else
75  /* 用户应用入口 */
76  int rt_application_init()
77  {
78          timer_control_init();
79
80          return 0;
81  }
82  #endif
```

CHAPTER

ELEVEN

I/O设备管理

I/O管理模块为应用提供了一个对设备进行访问的通用接口，并通过定义的数据结构对设备驱动程序和设备信息进行管理。从系统整体位置来说I/O管理模块相当于设备驱动程序和上层应用之间的一个中间层。

I/O管理模块实现了对设备驱动程序的封装：设备驱动程序的实现与I/O管理模块独立，提高了模块的可移植性。应用程序通过I/O管理模块提供的标准接口访问底层设备，设备驱动程序的升级不会对上层应用产生影响。这种方式使得与设备的硬件操作相关的代码与应用相隔离，双方只需各自关注自己的功能，这降低了代码的复杂性，提高了系统的可靠性。

在第5章中已经介绍过RT-Thread的内核对象管理器。读者若对这部分还不太了解，可以回顾一下这章。在RT-Thread中，设备也被认为是一类对象，被纳入对象管理器范畴。每个设备对象都是由基对象派生而来，每个具体设备都可以继承其父类对象的属性，并派生出其私有属性。下图即为设备对象的继承和派生关系示意图。

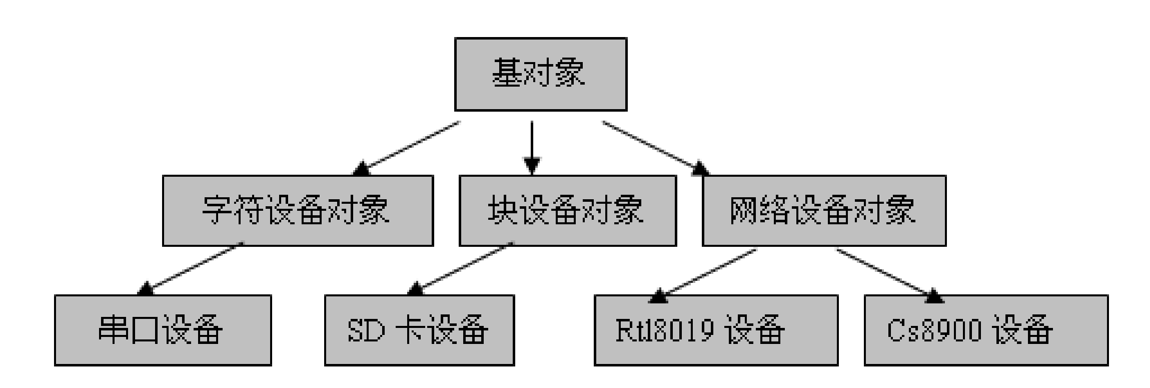


Figure 11.1: 设备对象的继承和派生关系示意图

11.1 I/O设备管理控制块

```
struct rt_device
{
    struct rt_object parent;

    /* 设备类型 */
```

```
    enum rt_device_class_type type;
    /* 设备参数及打开时参数 */
    rt_uint16_t flag, open_flag;

    /* 设备回调接口 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void* buffer);

    /* 设备公共接口 */
    rt_err_t  (*init)   (rt_device_t dev);
    rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
    rt_err_t  (*close)  (rt_device_t dev);
    rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
    rt_size_t (*write)  (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size);
    rt_err_t  (*control)(rt_device_t dev, rt_uint8_t cmd, void *args);

#ifdef RT_USING_DEVICE_SUSPEND
    rt_err_t (*suspend) (rt_device_t dev);
    rt_err_t (*resumed) (rt_device_t dev);
#endif

    /* 设备私有数据 */
    void* private;
};
```

当前RT-Thread支持的设备类型包括:

```
enum rt_device_class_type
{
        RT_Device_Class_Char = 0,       /* 字符设备     */
        RT_Device_Class_Block,          /* 块设备       */
        RT_Device_Class_NetIf,          /* 网络接口设备 */
        RT_Device_Class_MTD,            /* 内存设备     */
        RT_Device_Class_CAN,            /* CAN设备      */
        RT_Device_Class_Unknown         /* 未知设备     */
};
```

Note: suspend、resume回调函数只会在"RT_USING_DEVICE_SUSPEND"宏使能的情况下才会有效。

11.2 I/O设备管理接口

11.2.1 注册设备

在一个设备能够被上层应用访问前，需要先把这个设备注册到系统中，并添加一些相应的属性。这些注册的设备均可以采用"查找设备接口"通过设备名来查找设备，获得该设备控制块。注册设备的原始如下:

```
rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags)
```

其中调用的flags参数支持如下列表中的参数(可以采用或的方式支持多种参数):

```
#define RT_DEVICE_FLAG_DEACTIVATE       0x000   /* 未初始化设备                  */

#define RT_DEVICE_FLAG_RDONLY           0x001   /* 只读设备                      */
#define RT_DEVICE_FLAG_WRONLY           0x002   /* 只写设备                      */
#define RT_DEVICE_FLAG_RDWR             0x003   /* 读写设备                      */

#define RT_DEVICE_FLAG_REMOVABLE        0x004   /* 可移除设备                    */
#define RT_DEVICE_FLAG_STANDALONE       0x008   /* 独立设备                      */
#define RT_DEVICE_FLAG_ACTIVATED        0x010   /* 已激活设备                    */
#define RT_DEVICE_FLAG_SUSPENDED        0x020   /* 挂起设备                      */
#define RT_DEVICE_FLAG_STREAM           0x040   /* 设备处于流模式                */

#define RT_DEVICE_FLAG_INT_RX           0x100   /* 设备处于中断接收模式          */
#define RT_DEVICE_FLAG_DMA_RX           0x200   /* 设备处于DMA接收模式           */
#define RT_DEVICE_FLAG_INT_TX           0x400   /* 设备处于中断发送模式          */
#define RT_DEVICE_FLAG_DMA_TX           0x800   /* 设备处于DMA发送模式           */
```

RT_DEVICE_FLAG_STREAM参数用于向串口终端输出字符串，当输出的字符是"n"时，自动在前面补一个"r"做分行。

11.2.2 卸载设备

将设备从设备系统中卸载，被卸载的设备将不能通过“查找设备接口”找到该设备，可以通过如下接口完成：

```
rt_err_t rt_device_unregister(rt_device_t dev)
```

Note: 卸载设备并不会释放设备控制块所占用的内存。

11.2.3 初始化所有设备

初始化所有注册到设备对象管理器中的未初始化的设备，可以通过如下接口完成：

```
rt_err_t rt_device_init_all(void)
```

Note: 如果设备的flags域已经是RT_DEVICE_FLAG_ACTIVATED，调用这个接口将不再重复做初始化，一个设备初始化完成后它的flags域RT_DEVICE_FLAG_ACTIVATED应该被置位。

11.2.4 查找设备

根据指定的设备名称来查找设备，可以通过如下接口完成：

```
rt_device_t rt_device_find(const char* name)
```

使用以上接口时，在设备对象类型所对应的对象容器中遍历寻找设备对象，然后返回该设备，如果没有找到相应的设备对象，则返回RT_NULL。

11.2.5 打开设备

根据设备控制块来打开设备，可以通过如下接口完成：

```
rt_err_t rt_device_open (rt_device_t dev, rt_uint16_t oflags)
```

其中oflags支持以下列表中的参数：

```
#define RT_DEVICE_OFLAG_RDONLY      0x001   /* 只读模式访问     */
#define RT_DEVICE_OFLAG_WRONLY      0x002   /* 只写模式访问     */
#define RT_DEVICE_OFLAG_RDWR        0x003   /* 读写模式访问     */
```

Note: 如果设备flags域包含RT_DEVICE_FLAG_STANDALONE参数，将不允许重复打开。

11.2.6 关闭设备

根据设备控制块来关闭设备，可以通过如下接口完成：

```
rt_err_t rt_device_close(rt_device_t dev)
```

11.2.7 读设备

根据设备控制块来读取设备，可以通过如下接口完成：

```
rt_size_t rt_device_read (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
```

根据底层驱动的实现，通常这个接口并不会阻塞上层应用线程。返回值是读到数据的大小(以字节为单位)，如果返回值是0，需要读取当前线程的errno来判断错误状态。

11.2.8 写设备

根据设备控制块来写入设备，可以通过如下接口完成：

```
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)
```

根据底层驱动的实现，通常这个接口并不会阻塞上层应用线程。返回值是写入数据的大小(以字节为单位)，如果返回值是0，需要读取当前线程的errno来判断错误状态。

11.2.9 控制设备

根据设备控制块来控制设备，可以通过如下接口完成：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)
```

cmd命令参数通常是和设备驱动程序相关的。

11.2.10 设置数据接收指示

设置一个回调函数，当硬件设备收到数据时回调给应用程序以通知有数据达到。可以通过如下接口完成设置接收指示：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind )(rt_device_t dev,
            rt_size_t size))
```

回调函数rx_ind由调用者提供，当硬件设备接收到数据时，会回调这个函数并把收到的数据长度放在size参数中传递给上层应用。上层应用线程应在收到指示时，立刻从设备中读取数据。

11.2.11 设置发送完成指示

在上层应用调用rt_device_write写入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件给出发送完成时(例如DMA传送完成或FIFO已经写入完毕产生完成中断时)被调用。可以通过如下接口完成设备发送完成指示：

```
rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev,
            void *buffer))
```

回调函数tx_done由调用者提供，当硬件设备发送完数据时，由驱动程序回调这个函数并把发送完成的数据块地址buffer做为参数传递给上层应用。上层应用（线程）在收到指示时应根据发送buffer的情况，释放buffer内存块或为下一个写数据做缓存。

11.3 设备驱动

上一节说到了如何使用RT-Thread的设备接口，但对于开发人员来说，如何编写一个驱动设备可能会更加重要。

11.3.1 设备驱动必须实现的接口

我们先来看看/解析下RT-Thread的设备控制块：

```
struct rt_device
{
    struct rt_object parent;

    /* 设备类型 */
    enum rt_device_class_type type;
    /* 设备参数及打开时的参数 */
    rt_uint16_t flag, open_flag;

    /* 设备回调函数 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void* buffer);

    /* 公共的设备接口 */
    rt_err_t  (*init)   (rt_device_t dev);
    rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
```

```
    rt_err_t  (*close)  (rt_device_t dev);
    rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
    rt_size_t (*write)  (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size);
    rt_err_t  (*control)(rt_device_t dev, rt_uint8_t cmd, void *args);

            /* 当使用了设备挂起功能时的接口 */
#ifdef RT_USING_DEVICE_SUSPEND
    rt_err_t (*suspend) (rt_device_t dev);
    rt_err_t (*resumed) (rt_device_t dev);
#endif

    /* device private data */
    void* private;
};
```

其中包含了一个套公共的设备接口(类似上节说的设备访问接口，但面向的层次已经不一样了)：

```
/* 公共的设备接口 */
rt_err_t  (*init)   (rt_device_t dev);
rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
rt_err_t  (*close)  (rt_device_t dev);
rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
rt_size_t (*write)  (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size);
rt_err_t  (*control)(rt_device_t dev, rt_uint8_t cmd, void *args);

    /* 当使用了设备挂起功能时的接口 */
#ifdef RT_USING_DEVICE_SUSPEND
rt_err_t (*suspend) (rt_device_t dev);
rt_err_t (*resumed) (rt_device_t dev);
#endif
```

这些接口也是上层应用通过RT-Thread设备接口进行访问的实际底层接口，在满足一定的条件下，都会调用到这套接口。其中suspend和resume接口是应用于RT-Thread的电源管理部分，目前的0.3.0版本并不支持，预留给以后使用。

其他的六个接口，可以看成是底层设备驱动必须提供的接口。

Name	Description
init	设备的初始化。设备初始化完成后，设备控制块的flag会被置成已激活状态(RT_DEVICE_FLAG_ACTIVATED)。如果设备控制块的flag不是已激活状态，那么在设备框架调用rt_device_init_all接口时调用此设备驱动的init接口进行设备初始化。
open	打开设备。有些设备并不是系统一启动就已经打开开始运行的，或者设备需要进行数据接收，但如果上层应用还未准备好，设备也不应默认已经使能开始接收数据。所以建议底层驱动程序，在调用open接口时进行设备的使能。
close	关闭设备。在打开设备时，设备框架中会自动进行打开计数(设备控制块中的ocount数据域)，只有当打开计数为零的时候，底层设备驱动的close接口才会被调用。
read	从设备中读取数据。参数pos指出读取数据的偏移量，但是有些设备并不一定需要制定偏移量，例如串口设备，那么忽略这个参数即可。这个接口返回的类型是rt_size_t即读到的字节数，如果返回零建议检查errno值。如果errno值并不是RT_EOK，那么或者已经读取了所有数据，或者有错误发生。
write	往设备中写入数据。同样pos参数在一些情况下是不必要的，略过即可。
control	根据不同的cmd命令控制设备。命令往往是由底层设备驱动自定义实现的。

11.3.2 设备驱动实现的步骤

上节中比较详细介绍了RT-Thread的设备控制块，那么实现一个设备驱动的步骤是如何的：

1. 实现RT-Thread中定义的设备公共接口，开始可以是空函数(返回类型是rt_err_t的可默认返回RT_EOK)。
2. 根据自己的设备类型定义自己的私有数据域。特别是可以有多个相同设备的情况下，设备接口可以用同一套，不同的只是各自的数据域(例如寄存器基地址)。
3. 按照RT-Thread的对象模型，扩展一个对象有两种方式：
   (a) 定义自己的私有数据结构，然后赋值到RT-Thread设备控制块的private指针上。
   (b) 从struct rt_device结构中进行派生。
4. 根据设备的类型，注册到RT-Thread设备框架中。

Note: 异步设备，通俗的说就是，对设备进行读写并不是采用轮询的方式的，而是采用中断方式。例如接收中断产生代表接收到数据，发送中断产生代表数据已经真实的发送完毕。

11.3.3 AT91SAM7S64串口驱动

做为一个例子，这里仔细分析了AT91SAM7S64的串口驱动，也包括上层应该如何使用这个设备的代码。

AT91SAM7S64串口驱动代码，详细的中文注释已经放在其中了。

```
#include <rthw.h>
#include <rtthread.h>

#include "AT91SAM7X.h"
#include "serial.h"

/* 串口寄存器结构 */
```

```
typedef volatile rt_uint32_t REG32;
struct rt_at91serial_hw
{
    REG32    US_CR;     // Control Register
    REG32    US_MR;     // Mode Register
    REG32    US_IER;    // Interrupt Enable Register
    REG32    US_IDR;    // Interrupt Disable Register
    REG32    US_IMR;    // Interrupt Mask Register
    REG32    US_CSR;    // Channel Status Register
    REG32    US_RHR;    // Receiver Holding Register
    REG32    US_THR;    // Transmitter Holding Register
    REG32    US_BRGR;   // Baud Rate Generator Register
    REG32    US_RTOR;   // Receiver Time-out Register
    REG32    US_TTGR;   // Transmitter Time-guard Register
    REG32    Reserved0[5];  //
    REG32    US_FIDI;   // FI_DI_Ratio Register
    REG32    US_NER;    // Nb Errors Register
    REG32    Reserved1[1];  //
    REG32    US_IF;     // IRDA_FILTER Register
    REG32    Reserved2[44];     //
    REG32    US_RPR;    // Receive Pointer Register
    REG32    US_RCR;    // Receive Counter Register
    REG32    US_TPR;    // Transmit Pointer Register
    REG32    US_TCR;    // Transmit Counter Register
    REG32    US_RNPR;   // Receive Next Pointer Register
    REG32    US_RNCR;   // Receive Next Counter Register
    REG32    US_TNPR;   // Transmit Next Pointer Register
    REG32    US_TNCR;   // Transmit Next Counter Register
    REG32    US_PTCR;   // PDC Transfer Control Register
    REG32    US_PTSR;   // PDC Transfer Status Register
};

/* AT91串口设备 */
struct rt_at91serial
{
    /* 采用从设备基类中继承 */
    struct rt_device parent;

    /* 串口设备的私有数据 */
    struct rt_at91serial_hw* hw_base;   /* 寄存器基地址 */
    rt_uint16_t peripheral_id;          /* 外设ID */
    rt_uint32_t baudrate;               /* 波特率 */

    /* 用于接收的域 */
    rt_uint16_t save_index, read_index;
    rt_uint8_t  rx_buffer[RT_UART_RX_BUFFER_SIZE];
};
/* 串口类的实例化，serial 1/2 */
#ifdef RT_USING_UART1
struct rt_at91serial serial1;
#endif
#ifdef RT_USING_UART2
struct rt_at91serial serial2;
```

```
#endif

/* 串口外设的中断服务例程 */
static void rt_hw_serial_isr(int irqno)
{
    rt_base_t level;
    struct rt_device* device;
    struct rt_at91serial* serial = RT_NULL;

    /* 确定对应的外设对象 */
#ifdef RT_USING_UART1
    if (irqno == AT91C_ID_US0)
    {
        /* serial 1 */
        serial = &serial1;
    }
#endif
#ifdef RT_USING_UART2
    if (irqno == AT91C_ID_US1)
    {
        /* serial 2 */
        serial = &serial2;
    }
#endif
    RT_ASSERT(serial != RT_NULL);

    /* 获得设备基类对象指针 */
    device = (rt_device_t)serial;

    /* 关闭中断以更新接收缓冲 */
    level = rt_hw_interrupt_disable();

    /* 读取一个字符 */
    serial->rx_buffer[serial->save_index] = serial->hw_base->US_RHR;

    /* 把存放索引移到下一个位置 */
    serial->save_index ++;
    if (serial->save_index >= RT_UART_RX_BUFFER_SIZE)
        serial->save_index = 0;

    /* 如果存放索引指向的位置已经到了读索引位置，则丢掉一个数据 */
    if (serial->save_index == serial->read_index)
    {
        serial->read_index ++;
        if (serial->read_index >= RT_UART_RX_BUFFER_SIZE)
            serial->read_index = 0;
    }

    /* 使能中断 */
    rt_hw_interrupt_enable(level);

    /* 调用回调函数指示给上层收到了数据 */
    if (device->rx_indicate != RT_NULL)
```

```
        device->rx_indicate(device, 1);
}

/* 以下的是设备基类的公共接口(虚拟函数) */
static rt_err_t rt_serial_init (rt_device_t dev)
{
    rt_uint32_t bd;
    struct rt_at91serial* serial = (struct rt_at91serial*) dev;

    RT_ASSERT(serial != RT_NULL);
    /* 确认外设标识必需为US0或US1 */
    RT_ASSERT((serial->peripheral_id != AT91C_ID_US0) &&
              (serial->peripheral_id != AT91C_ID_US1));

    /* 使能时钟 */
    AT91C_PMC_PCER = 1 << serial->peripheral_id;

    /* 设置pinmux以使能Rx/Tx数据引脚 */
    if (serial->peripheral_id == AT91C_ID_US0)
    {
        AT91C_PIO_PDR = (1 << 5) | (1 << 6);
    }
    else if (serial->peripheral_id == AT91C_ID_US1)
    {
        AT91C_PIO_PDR = (1 << 21) | (1 << 22);
    }

    /* 重置外设 */
    serial->hw_base->US_CR = AT91C_US_RSTRX     |   /* Reset Receiver      */
                             AT91C_US_RSTTX     |   /* Reset Transmitter   */
                             AT91C_US_RXDIS     |   /* Receiver Disable    */
                             AT91C_US_TXDIS;        /* Transmitter Disable */

    /* 默认都设置为8-N-1 */
    serial->hw_base->US_MR = AT91C_US_USMODE_NORMAL |   /* Normal Mode */
                             AT91C_US_CLKS_CLOCK    |   /* Clock = MCK */
                             AT91C_US_CHRL_8_BITS   |   /* 8-bit Data  */
                             AT91C_US_PAR_NONE      |   /* No Parity   */
                             AT91C_US_NBSTOP_1_BIT;     /* 1 Stop Bit  */

    /* 设置波特率，注：主时钟（MCK）在board.h中定义 */
    bd =  ((MCK*10)/(serial->baudrate * 16));
    if ((bd % 10) >= 5) bd = (bd / 10) + 1;
    else bd /= 10;

    serial->hw_base->US_BRGR = bd;
    serial->hw_base->US_CR = AT91C_US_RXEN |    /* 使能接收 */
                             AT91C_US_TXEN;     /* 使能发送 */

    /* 重置读写索引 */
    serial->save_index = 0;
    serial->read_index = 0;
```

```
    /* 重置接收缓冲 */
    rt_memset(serial->rx_buffer, 0, RT_UART_RX_BUFFER_SIZE);

    return RT_EOK;
}

static rt_err_t rt_serial_open(rt_device_t dev, rt_uint16_t oflag)
{
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    /* 如果是中断方式接收，打开中断并装载中断 */
    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* enable UART rx interrupt */
        serial->hw_base->US_IER = 1 << 0;        /* 使能RxReady中断 */
        serial->hw_base->US_IMR |= 1 << 0;       /* 激活RxReady中断 */

        /* 转载UART中断服务例程 */
        rt_hw_interrupt_install(serial->peripheral_id, rt_hw_serial_isr, RT_NULL);
        AT91C_AIC_SMR(serial->peripheral_id) = 5 | (0x01 << 5);
        rt_hw_interrupt_umask(serial->peripheral_id);
    }

    return RT_EOK;
}

static rt_err_t rt_serial_close(rt_device_t dev)
{
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    /* 如果是中断方式接收，关闭中断 */
    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* disable interrupt */
        serial->hw_base->US_IDR = 1 << 0;        /* 关闭RxReady中断 */
        serial->hw_base->US_IMR &= ~(1 << 0);    /* 屏蔽RxReady中断 */
    }

    /* 重置外设 */
    serial->hw_base->US_CR = AT91C_US_RSTRX     |   /* Reset Receiver      */
                             AT91C_US_RSTTX     |   /* Reset Transmitter   */
                             AT91C_US_RXDIS     |   /* Receiver Disable    */
                             AT91C_US_TXDIS;        /* Transmitter Disable */

    return RT_EOK;
}

static rt_size_t rt_serial_read (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
{
    rt_uint8_t* ptr;
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
```

```
    RT_ASSERT(serial != RT_NULL);

    /* ptr指向读取的缓冲 */
    ptr = (rt_uint8_t*) buffer;

    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* 中断模式接收 */
        while (size)
        {
            rt_base_t level;

            /* serail->rx_buffer是和ISR共享的，需要关闭中断保护 */
            level = rt_hw_interrupt_disable();
            if (serial->read_index != serial->save_index)
            {
                *ptr = serial->rx_buffer[serial->read_index];

                serial->read_index ++;
                if (serial->read_index >= RT_UART_RX_BUFFER_SIZE)
                    serial->read_index = 0;
            }
            else
            {
                /* rx buffer中无数据 */

                /* 使能中断 */
                rt_hw_interrupt_enable(level);
                break;
            }

            /* 使能中断 */
            rt_hw_interrupt_enable(level);

            ptr ++;
            size --;
        }

        return (rt_uint32_t)ptr - (rt_uint32_t)buffer;
    }
    else if (dev->flag & RT_DEVICE_FLAG_DMA_RX)
    {
        /* DMA模式接收，目前不支持 */
        RT_ASSERT(0);
    }
    else
    {
        /* 轮询模式 */
        while (size)
        {
            /* 等待数据达到 */
            while (!(serial->hw_base->US_CSR & AT91C_US_RXRDY));
```

```
            /* 读取一个数据 */
            *ptr = serial->hw_base->US_RHR;
            ptr ++;
            size --;
        }

        return (rt_size_t)ptr - (rt_size_t)buffer;
    }

    return 0;
}

static rt_size_t rt_serial_write (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)
{
    rt_uint8_t* ptr;
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    ptr = (rt_uint8_t*) buffer;
    if (dev->open_flag & RT_DEVICE_OFLAG_WRONLY)
    {
        if (dev->flag & RT_DEVICE_FLAG_STREAM)
        {
            /* STREAM模式发送 */
            while (size)
            {
                /* 遇到'\n'进入STREAM模式， 在前面添加一个'\r' */
                if (*ptr == '\n')
                {
                    while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));
                    serial->hw_base->US_THR = '\r';
                }

                /* 等待发送就绪 */
                while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));

                /* 发送单个字符 */
                serial->hw_base->US_THR = *ptr;
                ptr ++;
                size --;
            }
        }
        else
        {
            while (size)
            {
                /* 等待发送就绪 */
                while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));

                /* 发送单个字符 */
                serial->hw_base->US_THR = *ptr;
                ptr ++;
                size --;
```

```
            }
        }
    }

    return (rt_size_t)ptr - (rt_size_t)buffer;
}

static rt_err_t rt_serial_control (rt_device_t dev, rt_uint8_t cmd, void *args)
{
    return RT_EOK;
}

/* 串口设备硬件初始化，它会根据配置情况进行串口设备注册 */
rt_err_t rt_hw_serial_init()
{
    rt_device_t device;

#ifdef RT_USING_UART1
    device = (rt_device_t) &serial1;

    /* 初始化AT91串口设备私有数据 */
    serial1.hw_base          = (struct rt_at91serial_hw*)AT91C_BASE_US0;
    serial1.peripheral_id    = AT91C_ID_US0;
    serial1.baudrate         = 115200;

    /* 设置设备基类的虚拟函数接口 */
    device->init    = rt_serial_init;
    device->open    = rt_serial_open;
    device->close   = rt_serial_close;
    device->read    = rt_serial_read;
    device->write   = rt_serial_write;
    device->control = rt_serial_control;

    /* 在设备子系统中注册uart1设备 */
    rt_device_register(device, "uart1", RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX);
#endif

#ifdef RT_USING_UART2
    /* 初始化AT91串口设备私有数据 */
    device = (rt_device_t) &serial2;

    serial2.hw_base          = (struct rt_at91serial_hw*)AT91C_BASE_US1;
    serial2.peripheral_id    = AT91C_ID_US1;
    serial2.baudrate         = 115200;

    /* 设置设备基类的虚拟函数接口 */
    device->init    = rt_serial_init;
    device->open    = rt_serial_open;
    device->close   = rt_serial_close;
    device->read    = rt_serial_read;
    device->write   = rt_serial_write;
    device->control = rt_serial_control;
```

```
    /* 在设备子系统中注册uart2设备 */
    rt_device_register(device, "uart2", RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX);
#endif

    return RT_EOK;
}
```

这个驱动程序中是包含中断发送、接收的情况，所以针对这些，下面给出了具体的使用代码。在这个例子中，线程将在两个设备上(UART1, UART2)读取数据，然后再写到其中的一个设备中。

```
#include <rtthread.h>

/* UART接收消息结构 */
struct rx_msg
{
    rt_device_t dev;
    rt_size_t size;
};

/* 用于接收消息的消息队列 */
static rt_mq_t rx_mq;
/* 接收线程的接收缓冲区 */
static char uart_rx_buffer[64];

/* 数据达到回调函数 */
rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    struct rx_msg msg;
    msg.dev  = dev;
    msg.size = size;

    /* 发送消息到消息队列中 */
    rt_mq_send(rx_mq, &msg, sizeof(struct rx_msg));

    return RT_EOK;
}

void device_thread_entry(void* parameter)
{
    struct rx_msg msg;
    int count = 0;

    rt_device_t device, write_device;
    rt_err_t result = RT_EOK;

    device = rt_device_find("uart1");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备 */
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR);
    }
```

```
    /* 设置写设备 */
    write_device = device;
    device = rt_device_find("uart2");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备 */
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR);
    }

    while (1)
    {
        /* 从消息队列中读取消息 */
        result = rt_mq_recv(rx_mq, &msg, sizeof(struct rx_msg), 50);
        if (result == -RT_ETIMEOUT)
        {
            /* 接收超时 */
            rt_kprintf("timeout count:%d\n", ++count);
        }

        /* 成功收到消息 */
        if (result == RT_EOK)
        {
            rt_uint32_t rx_length;

            rx_length = (sizeof(uart_rx_buffer) - 1) > msg.size ?
                msg.size : sizeof(uart_rx_buffer) - 1;

            /* 读取消息 */
            rx_length = rt_device_read(msg.dev, 0, &uart_rx_buffer[0], rx_length);
            uart_rx_buffer[rx_length] = '\0';

            /* 写到写设备中 */
            if (write_device != RT_NULL)
                rt_device_write(write_device, 0, &uart_rx_buffer[0], rx_length);
        }
    }
}

int rt_application_init()
{
    /* 创建devt线程 */
    rt_thread_t thread = rt_thread_create("devt",
        device_thread_entry, RT_NULL,
        1024, 25, 7);

    /* 创建成功则启动线程 */
    if (thread != RT_NULL)
        rt_thread_startup(&thread);
}
```

线程devt启动后，将先查找是否有存在uart1, uart2两个设备，如果存在则设置数据接收到回调函数。在数据接收到的回调函数中，将把对应的设备句柄，接收到的数据长度填充到一个消息结构

（struct rx_msg）上，然后发送到消息队列中。devt线程在打开完设备后，将在消息队列中等待消息的到来。如果消息队列是空的，devt线程将被阻塞，直到它接收到消息被唤醒，或在0.5秒(50 OS Tick)内都没收到消息而唤醒。两者唤醒时，从rt_mq_recv函数的返回值中是不相同的。当devt线程因为接收到消息而唤醒时（rt_mq_recv函数的返回值是RT_EOK），它将主动调用rt_device_read去读取消息，然后写入到write_device设备中。

CHAPTER

TWELVE

FINSH SHELL系统

RT-Thread的shell系统——finsh，提供了一套供用户在命令行操作的接口，主要用于调试、查看系统信息。finsh被设计成一个不同于传统命令行的C语言表达式解释器：由于很多嵌入式系统都是采用C语言来编写，finsh正是采用了这种系统软件开发人员都会的语法形式，把C语言表达式变成了命令行的风格。它能够解析执行大部分C语言的表达式，也能够使用类似于C语言的函数调用方式访问系统中的函数及全局变量，此外它也能够通过命令行方式创建变量。

12.1 基本数据类型

finsh支持基本的C语言数据类型，包括：

Data Type	Description
void	空数据格式，只用于创建指针变量
char, unsigned char	(带符号)字符型数据
short, unsigned int	(带符号)整数型数据
long, unsigned long	(带符号)长整型数据

此外，finsh也支持指针类型（void *或int *等声明方式），如果指针做为函数指针类型调用，将自动按照函数方式执行。

finsh中内建了一些命令函数，可以在命令行中调用:

```
list()
```

Warning: 在finsh shell中使用命令（即C语言中的函数），必须类似C语言中的函数调用方式，即必须携带“()”符号。而最后finsh shell的输出为此函数的返回值。对于一些不存在返回值的函数，这个打印输出没有意义。

显示系统中存在的命令及变量，在AT91SAM7S64平台上执行结果如下:

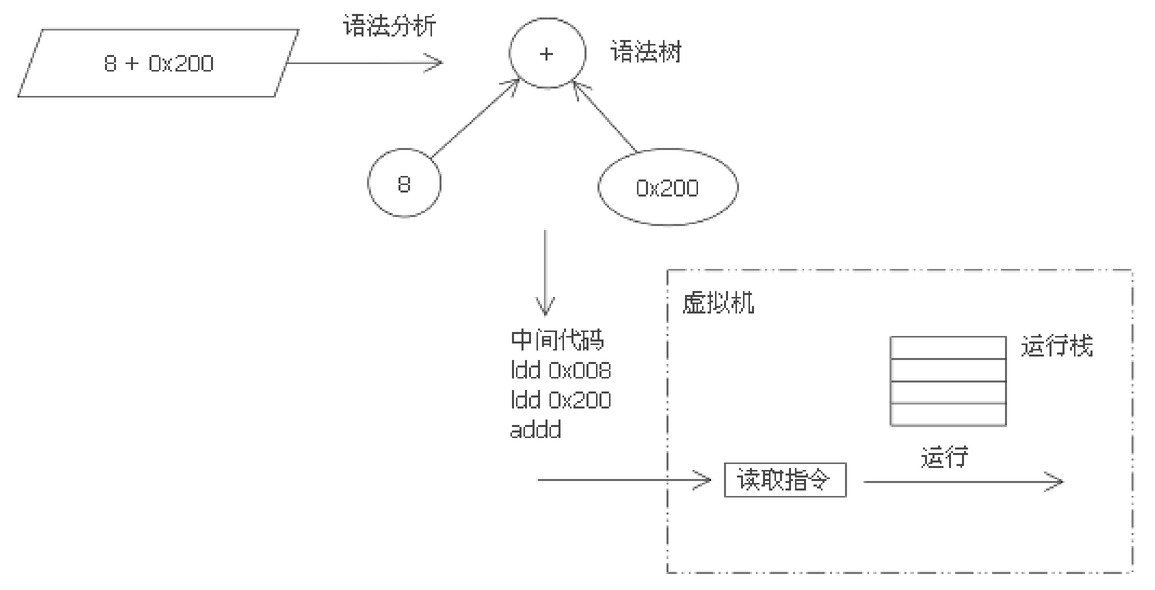
```
--Function List:
hello
version
list
list_thread
list_sem
list_mutex
list_event
```

```
list_mb
list_mq
list_memp
list_timer
--Variable List:
```

–Function List表示的是函数列表； –Variable List表示的是变量列表。

12.2 工作模式

FinSH shell的实现采用了完整的语法分析，文法分析，中间代码生成，虚拟机运行等编译器技术，其分析流程如下：



当finsh shell接收到例如 8 + 0x200 的命令输入时，它将根据预定的语法规则对它进行扫描，形成一颗语法树。然后FinSH将进入编译的阶段，扫描语法树上所有节点，编译成相应的中间码(finsh虚拟机的机器指令)。finsh虚拟机被实现成一个基于栈的虚拟机，它将从编译出来的中间码中依次读取指令，例如ldd 0x08指令，它将把0x08载入到机器栈中；ldd 0x200指令，它将把0x200载入到机器栈中；addd指令，虚拟机会从栈中读取栈顶的两个值进行相加操作，然后把结果放到栈中。

此外虚拟机也包括，一个字符串池，用于放置运行过程中使用到的字符串（在指令中依据地址对它进行引用）；一个系统变量区，用于放置创建了的变量。

12.3 RT-Thread内置命令

针对RT-Thread RTOS，finsh提供了一系列基本的函数命令：

12.3.1 list_thread()

列表显示当前系统中线程状态，会显示类似如下的结果:

```
thread   pri  status      sp     stack size max used   left tick  error
-------- ---- ------- ---------- ---------- ---------- ---------- ---
tidle    0xff ready   0x00000074 0x00000100 0x00000074 0x0000003b 000
tshell   0x14 ready   0x0000024c 0x00000800 0x00000418 0x00000064 000

thread字段表示线程的名称
pri字段表示线程的优先级
status字段表示线程当前的状态
sp字段表示线程当前的栈位置
stack size字段表示线程的栈大小
max used字段表示线程历史上使用的最大栈位置
left tick字段表示线程剩余的运行节拍数
error字段表示线程的错误号
```

12.3.2 list_sem()

列表显示系统中信号量状态，会显示类似如下的结果:

```
semaphore v   suspend thread
--------  --- --------------
uart      000 0

semaphore字段表示信号量的名称
v字段表示信号量的当前值
suspend thread字段表示等在这个信号量上的线程数目
```

12.3.3 list_mb()

列表显示系统中信箱状态，会显示类似如下的结果:

```
mailbox  entry  size suspend thread
-------- -----  ---- --------------

mailbox字段表示信箱的名称
entry字段表示信箱中包含的信件数目
size字段表示信箱能够容纳的最大信件数目
suspend thread字段表示等在这个信箱上的线程数目
```

12.3.4 list_mq()

列表显示系统中消息队列状态，会显示类似如下的结果:

```
msgqueue entry  suspend thread
-------- -----  --------------

semaphore字段表示消息队列的名称
entry字段表示消息队列中当前包含的消息数目
size字段表示消息队列能够容纳的最大消息数目
suspend thread字段表示等在这个消息队列上的线程数目
```

12.3.5 list_event()

列表显示系统中事件状态，会显示类似如下的结果:

```
event    set        suspend thread
-------- ---------- --------------
```

```
event字段表示事件的名称
set字段表示事件的值
suspend thread字段表示等在这个事件上的线程数目
```

12.3.6 list_timer()

列表显示系统中定时器状态，会显示类似如下的结果:

```
timer    periodic   timeout    flag
-------- ---------- ---------- -----------
tidle    0x00000000 0x00000000 deactivated
tshell   0x00000000 0x00000000 deactivated
current tick:0x00000d7e
```

```
timer字段表示定时器的名称
periodic字段表示定时器是否是周期性的
timeout字段表示定时器超时时的节拍数
flag字段表示定时器的状态，activated表示活动的，deactivated表示不活动的
current tick表示当前系统的节拍
```

12.4 应用程序接口

finsh的应用程序接口提供了上层注册函数或变量的接口，使用时应包含如下头文件：

```
#include <finsh.h>
```

注：另外一种添加函数及变量的方式，请参看本章选项一节。

12.4.1 添加函数

```
void finsh_syscall_append(const char* name, syscall_func func)
```

在finsh中添加一个函数。 name – 函数在finsh shell中访问的名称 func – 函数的地址

12.4.2 添加变量

```
void finsh_sysvar_append(const char* name, u_char type, void* addr)
```

这个接口用于在finsh中添加一个变量:

```
name - 变量在finsh shell中访问的名称
type - 数据类型，由枚举类型finsh_type给出。当前finsh支持的数据类型：

.. code-block:: c

    enum finsh_type {
        finsh_type_unknown = 0,
        finsh_type_void,                /** void                */
        finsh_type_voidp,               /** void pointer        */
        finsh_type_char,                /** char                */
        finsh_type_uchar,               /** unsigned char       */
        finsh_type_charp,               /** char pointer        */
        finsh_type_short,               /** short               */
        finsh_type_ushort,              /** unsigned short      */
        finsh_type_shortp,              /** short pointer       */
        finsh_type_int,                 /** int                 */
        finsh_type_uint,                /** unsigned int        */
        finsh_type_intp,                /** int pointer         */
        finsh_type_long,                /** long                */
        finsh_type_ulong,               /** unsigned long       */
        finsh_type_longp                /** long pointer        */
    };

addr - 变量的地址
```

12.4.3 宏方式输出函数、变量

当使能了FINSH_USING_SYMTAB宏时，也能够使用宏输出的方式向finsh shell增加命令。当需要输出函数或变量到finsh shell时，可以通过引用宏：`FINSH_FUNCTION_EXPORT`和 `FINSH_VAR_EXPORT`的方式。例如：

```
long hello()
{
    rt_kprintf("Hello RT-Thread!\n");

    return 0;
}
FINSH_FUNCTION_EXPORT(hello, say hello world)

static int dummy = 0;
FINSH_VAR_EXPORT(dummy, finsh_type_int, dummy variable for finsh)
```

hello函数、counter变量将自动输出到shell中，即可在shell中调用、访问hello函数、counter变量。而定义了宏`FINSH_USING_DESCRIPTION`将可以在list()列出函数、变量列表时显示相应的帮助描述。

12.5 移植

由于finsh完全采用ANSI C编写，具备极好的移植性，同时在内存占用上也非常小，如果不使用上述提到的API函数，整个finsh将不会动态申请内存。

- finsh shell线程：

每次的命令执行都是在finsh shell线程的上下文中完成的，finsh shell线程在函数finsh_system_init()中创建，它将一直等待uart_sem信号量的释放。

- finsh的输出：

finsh的输出依赖于系统的输出，在RT-Thread中依赖的是rt_kprintf输出。

- finsh的输入：

finsh shell线程在获得了uart_sem信号量后调用rt_serial_getc()函数从串口中获得一个字符然后处理。所以finsh的移植需要rt_serial_getc()函数的实现。而uart_sem信号量的释放通过调用finsh_notify()函数以完成对finsh shell线程的输入通知。

通常的过程是，当串口接收中断发生时（即串口中有输入），接收中断服务例程调用finsh_notify()函数通知finsh shell线程有输入；而后finsh shell线程获取串口输入最后做相应的命令处理。

12.6 选项

要开启finsh的支持，在RT-Thread的配置中必须定义RT_USING_FINSH宏。

```
/* SECTION: FinSH shell options */
/* 使用FinSH做为shell*/
#define RT_USING_FINSH
/* 使用内置符号表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION
```

CHAPTER

THIRTEEN

文件系统

RT-Thread的文件系统采用了三层的结构，如下图所示：

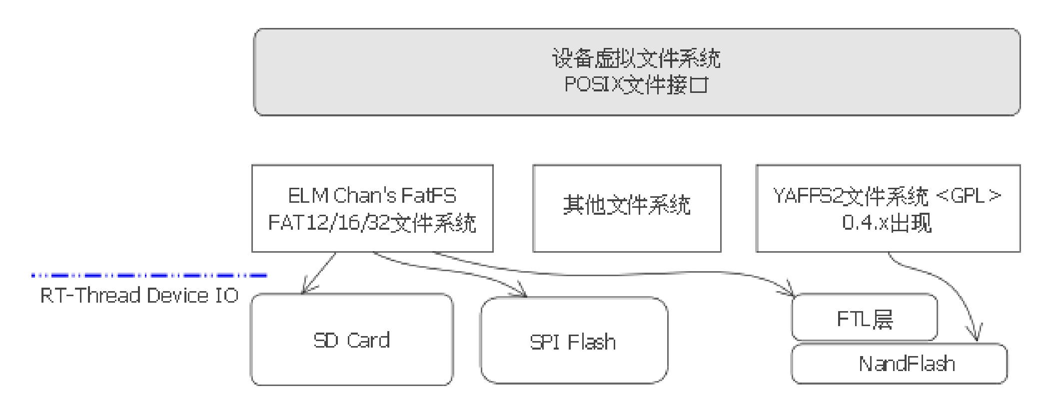


Figure 13.1: 文件系统结构

- 最顶层的是一套面向嵌入式系统专门优化过的虚拟文件系统（接口），通过它能够适配下层不同的文件系统格式，例如个人电脑上常使用的FAT文件系统，或者是嵌入式设备中常用的flash文件系统。
- 接下来的是各种文件系统的实现，例如支持FAT文件系统的DFS-ELM；支持NandFlash的YAFFS2将在0.4.x分支中移植进这套虚拟文件系统框架中。
- 最底层的是各类存储驱动，例如SD卡驱动，IDE硬盘驱动等。0.4.x分支上将在NandFlash上构建一层转换层(FTL)以使得NandFlash能够支持Flash文件系统。

RT-Thread的文件系统对上层提供的接口主要以POSIX标准接口为主，这样这部分代码也容易调试通过。

13.1 文件系统接口

13.1.1 打开文件

打开或创建一个文件可以调用如下接口

```
int open(const char *pathname, int oflag, int mode)
```

pathname是要打开或创建的文件名，oflag指出打开文件的选项，当前可以支持：

Name	Description
O_RDONLY	RD只读方式打开
O_WRONLY	WR只写方式打开
O_RDWR	RD-WR读写方式打开
O_CREAT	如果要打开的文件不存在，则建立该文件
O_APPEND	当读写文件时会从文件尾开始移动，也就是所定入的数据会以附加的方式加入到文件后面
O_DIRECTORY	如果参数pathname所指的文件并非为一个目录，则会令打开文件失败

mode是为了保持和POSIX标准接口像兼容的参数，目前没有意义，传入0即可。

打开成功时返回打开文件的描述符序号

```
/* 假设文件操作是在一个线程中完成 */
void file_thread()
{
    int fd, size;
    char s[] = "RT-Thread Programmer!\n", buffer[80];

    /* 打开 /text.txt 作写入，如果该文件不存在则建立该文件 */
    fd = open("/text.txt", O_WRONLY | O_CREAT);
    if (fd >= 0)
    {
        write(fd, s, sizeof(s));
        close(fd);
    }

    /* 打开 /text.txt 准备作读取动作 */
    fd = open("/text.txt", O_RDONLY);
    if (fd >= 0)
    {
        size=read(fd, buffer, sizeof(buffer));
        close(fd);
    }

    printf("%s", buffer);
}
```

13.1.2 关闭文件

```
int close(int fd)
```

当使用完文件后若已不再需要则可使用close()关闭该文件，而close()会让数据写回磁盘，并释放该文件所占用的资源。参数fd为先前由open()或creat()所返回的文件描述词。

13.1.3 读取数据

```
ssize_t read(int fd, void *buf, size_t count)
```

read()函数会把参数fd所指的文件传送count个字节到buf指针所指的内存中。若参数count为0，则read()不会有作用并返回0。返回值为实际读取到的字节数，如果返回0，表示已到达文件尾或是无可读取的数据，此外文件读写位置随读取到的字节移动。

13.1.4 写入数据

```
size_t write(int fd, const void *buf, size_t count)
```

write()会把参数buf所指的内存写入count个字节到参数fd所指的文件内。当然，文件读写位置也会随之移动。如果顺利write()会返回实际写入的字节数。当有错误发生时则返回-1，错误代码存入当前线程的errno中。

一个完整的文件读写例程：

```
/*
 * 代码清单：文件读写例子
 *
 * 这个例子演示了如何读写一个文件，特别是写的时候应该如何操作。
 */

#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

#define TEST_FN                 "/test.dat"

/* 测试用的数据和缓冲 */
static char test_data[120], buffer[120];

/* 文件读写测试 */
void readwrite(const char* filename)
{
        int fd;
        int index, length;

        /* 只写 & 创建 打开 */
        fd = open(TEST_FN, O_WRONLY | O_CREAT | O_TRUNC, 0);
        if (fd < 0)
        {
                rt_kprintf("open file for write failed\n");
                return;
        }

        /* 准备写入数据 */
        for (index = 0; index < sizeof(test_data); index ++)
        {
                test_data[index] = index + 27;
        }
```

```
34
35          /* 写入数据 */
36          length = write(fd, test_data, sizeof(test_data));
37          if (length != sizeof(test_data))
38          {
39                  rt_kprintf("write data failed\n");
40                  close(fd);
41                  return;
42          }
43
44          /* 关闭文件 */
45          close(fd);
46
47          /* 只写并在末尾添加打开 */
48          fd = open(TEST_FN, O_WRONLY | O_CREAT | O_APPEND, 0);
49          if (fd < 0)
50          {
51                  rt_kprintf("open file for append write failed\n");
52                  return;
53          }
54
55          length = write(fd, test_data, sizeof(test_data));
56          if (length != sizeof(test_data))
57          {
58                  rt_kprintf("append write data failed\n");
59                  close(fd);
60                  return;
61          }
62          /* 关闭文件 */
63          close(fd);
64
65          /* 只读打开进行数据校验 */
66          fd = open(TEST_FN, O_RDONLY, 0);
67          if (fd < 0)
68          {
69                  rt_kprintf("check: open file for read failed\n");
70                  return;
71          }
72
73          /* 读取数据(应该为第一次写入的数据) */
74          length = read(fd, buffer, sizeof(buffer));
75          if (length != sizeof(buffer))
76          {
77                  rt_kprintf("check: read file failed\n");
78                  close(fd);
79                  return;
80          }
81
82          /* 检查数据是否正确 */
83          for (index = 0; index < sizeof(test_data); index ++)
84          {
85                  if (test_data[index] != buffer[index])
86                  {
```

```
87                          rt_kprintf("check: check data failed at %d\n", index);
88                          close(fd);
89                          return;
90                  }
91          }
92
93          /* 读取数据(应该为第二次写入的数据) */
94          length = read(fd, buffer, sizeof(buffer));
95          if (length != sizeof(buffer))
96          {
97                  rt_kprintf("check: read file failed\n");
98                  close(fd);
99                  return;
100         }
101
102         /* 检查数据是否正确 */
103         for (index = 0; index < sizeof(test_data); index ++)
104         {
105                 if (test_data[index] != buffer[index])
106                 {
107                         rt_kprintf("check: check data failed at %d\n", index);
108                         close(fd);
109                         return;
110                 }
111         }
112
113         /* 检查数据完毕，关闭文件 */
114         close(fd);
115         /* 打印结果 */
116         rt_kprintf("read/write done.\n");
117 }
118
119 #ifdef RT_USING_FINSH
120 #include <finsh.h>
121 /* 输出函数到finsh shell命令行中 */
122 FINSH_FUNCTION_EXPORT(readwrite, perform file read and write test);
123 #endif
```

13.1.5 更改名称

```
int rename(const char *oldpath, const char *newpath)
```

rename()会将参数oldpath所指定的文件名称改为参数newpath所指的文件名称。或newpath所指定的文件已经存在，则会被删除。

例子代码如下：

```
void file_thread(void* parameter)
{
    rt_kprintf("%s => %s ", "/text1.txt", "/text2.txt");
```

```
    if(rename("/text1.txt", "/text2.txt") <0 )
     rt_kprintf("[error!]\n");
    else
     rt_kprintf("[ok!]\n");
}
```

13.1.6 取得状态

```
int  stat(const char *file_name, struct stat *buf)
```

stat()函数用来将参数file_name所指的文件状态，复制到参数buf所指的结构中(struct stat)。

例子如下：

```
void file_thread(void* parameter)
{
      struct stat buf;
      stat("/text.txt", &buf);

      rt_kprintf("text.txt file size = %d\n", buf.st_size);
}
```

13.2 目录操作接口

13.2.1 创建目录

```
int mkdir (const char *path, rt_uint16_t mode)
```

mkdir()函数用来创建一个目录，参数path为目录名，参数mode在当前版本未启用，输入0x777即可，若目录创建成功，返回0，否则返回-1.

例子如下：

```
void file_thread(void* parameter)
{
    int ret;

    /* 创建目录 */
    ret = mkdir("/web", 0x777)
    if(ret < 0)
    {
        /* 创建目录失败 */
        rt_kprintf("[mkdir error!]\n");
    }
    else
    {   /* 创建目录成功 */
        rt_kprintf("[mkdir ok!]\n");
    }
}
```

13.2.2 打开目录

```
DIR* opendir(const char* name)
```

opendir()函数用来打开一个目录，参数为目录路径名，若读取目录成功，返回该目录结构，若读取目录失败，返回RT_NULL。

例子如下：

```
void dir_operation(void* parameter)
{
    int result;
    DIR *dirp;

    /* 创建 /web 目录 */
    dirp = opendir("/web");
    if(dirp == RT_NULL)
    {
        rt_kprintf("[error!]\n");
    }
    else
    {
        /* 在这儿进行读取目录相关操作 */
        /* ...... */
    }
}
```

13.2.3 读取目录

```
struct dirent* readdir(DIR *d)
```

readdir()函数用来读取目录，参数为目录路径名，返回值为读到的目录项结构，如果返回值为RT_NULL，则表示已经读到目录尾。此外，没读取一次目录，目录流的指针位置为自动往后递推1个位置。

例子如下：

```
void dir_operation(void* parameter)
{
    int result;
    DIR *dirp;
    struct dirent *d

    /* 打开 /web 目录 */
    dirp = opendir("/web");
    if(dirp == RT_NULL)
    {
        rt_kprintf("[error!]\n");
    }
    else
    {
```

```
        /* 读取目录 */
        while ((d = readdir(dirp)) != RT_NULL)
        {
            rt_kprintf("found %s\n",d->d_name);
        }
    }
}
```

13.2.4 取得目录流的读取位置

```
rt_off_t telldir(DIR *d)
```

telldir()函数用来取得当前目录流的读取位置。

13.2.5 设置下次读取目录的位置

```
void seekdir(DIR *d, rt_off_t offset)
```

seekdir()函数用来设置下回目录读取的位置。

例子如下：

```
void dir_operation(void* parameter)
{
    DIR * dirp;
    int save3 = 0;
    int cur;
    int i = 0;
    struct dirent *dp;

    dirp = opendir (".");
    for (dp = readdir (dirp); dp != RT_NULL; dp = readdir (dirp))
    {
        /* 保存第三个目录项的目录指针 */
        if (i++ == 3)
        save3 = telldir (dirp);

        rt_kprintf ("%s\n", dp->d_name);
    }

    /* 回到刚才保存的第三个目录项的目录指针 */
    seekdir (dirp, save3);

    /* 检查当前目录指针是否等于保存过的第三个目录项的指针.  */
    cur = telldir (dirp);
    if (cur != save3)
    {
        rt_kprintf ("seekdir (d, %ld); telldir (d) == %ld\n", save3, cur);
    }
```

```
    /* 从第三个目录项开始打印 */
    for (dp = readdir (dirp); dp != NULL; dp = readdir (dirp))
    rt_kprintf ("%s\n", dp->d_name);

    /* 关闭目录 */
    closedir (dirp);
}
```

13.2.6 重设读取目录的位置为开头位置

```
void rewinddir(DIR *d)
```

rewinddir()函数用来设置读取的目录位置为开头位置。

13.2.7 关闭目录

```
int closedir(DIR* d)
```

closedir()函数用来关闭一个目录，如果关闭目录成功返回0，否则返回-1，该函数必须和opendir()函数成对出现。

13.2.8 删除目录

```
int rmdir(const char *pathname)
```

rmdir()函数用来删除一个目录，如果删除目录成功返回0，否则返回-1。

13.3 下层驱动接口

RT-Thread DFS文件系统针对下层媒介使用的是RT-Thread的设备系统，其中主要包括设备读写等操作。

13.4 文件系统初始化

在使用文件系统接口前，需要对文件系统进行初始化，代码包括：

```
/* 初始化线程 */
void rt_init_thread_entry(void *parameter)
{
    /* 文件系统初始化 */
#ifdef RT_USING_DFS
    {
    /* 初始化设备文件系统 */
    dfs_init();
```

```
#ifdef RT_USING_DFS_ELMFAT
    /* 如果使用的是ELM的FAT文件系统，需要对它进行初始化 */
    elm_init();

    /* 调用dfs_mount函数对设备进行挂接 */
    if (dfs_mount("sd0", "/", "elm", 0, 0) == 0)
        rt_kprintf("File System initialized!\n");
    else
        rt_kprintf("File System init failed!\n");
#endif
    }
#endif
}
```

其中主要包括的接口是，

```
int dfs_mount(const char* device_name, const char* path,
    const char* filesystemtype, rt_uint32_t rwflag, const
    void* data);
```

这个函数用于把以device_name为名称的设备挂接到path路径中，filesystemtype指定了文件系统的类型（如上面代码所述的efs或elm），然后是flag和data。

data对某些文件系统是有意义的，elm则不需要。

CHAPTER

FOURTEEN

TCP/IP协议栈

LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的Adam Dunkels等开发的一套用于嵌入式系统的开放源代码TCP/IP协议栈，它在包含完整的TCP协议实现基础上实现了小型的资源占用，因此它十分适合于使用到嵌入式设备中，占用的体积大概在几十kB RAM和40KB ROM代码左右。

由于 LwIP 出色的小巧实现，而功能也相对完善，用户群比较广泛，RT-Thread 采用 LwIP 做为默认的 TCP/IP 协议栈，同时根据小型设备的特点对其进行再优化，体积相对进一步减小，RAM 占用缩小到5kB附近（依据上层应用使用情况会有浮动）。本章主要讲述了 LwIP 在 RT-Thread 中的使用。

14.1 协议初始化

在使用 LwIP 协议栈之前，需要初始化协议栈。协议栈本身会启动一个 TCP 的线程，和协议相关的处理都会放在这个线程中完成。

```
#include <rtthread.h>

#ifdef RT_USING_LWIP
#include <lwip/sys.h>
    #include <netif/ethernetif.h>
#endif

/* 初始化线程入口 */
void rt_init_thread_entry(void *parameter)
{
/* LwIP 初始化 */
#ifdef RT_USING_LWIP
    {
        extern void lwip_sys_init(void);

                    /* 初始化以太网线程 */
                    eth_system_device_init();

                    /* 注册以太网接口驱动 */
                    rt_hw_stm32_eth_init();
                    /* 初始化注册的设备驱动，它仅会初始化未初始化的驱动 */
                    rt_device_init_all();
```

```
        /* 初始化LwIP系统 */
        lwip_sys_init();
        rt_kprintf("TCP/IP initialized!\n");
    }
#endif
}

int rt_application_init()
{
    rt_thread_t init_thread;

    /* 创建初始化线程 */
    init_thread = rt_thread_create("init",
        rt_init_thread_entry, RT_NULL,
        2048, 10, 5);
    /* 启动线程 */
    if (init_thread != RT_NULL) rt_thread_startup(init_thread);

    return 0;
}
```

另外，在RT-Thread中为了使用 LwIP 协议栈需要在 rtconfig.h 头文件中定义使用 LwIP的宏

```
/* 使用 lighweight TCP/IP 协议栈 */
#define RT_USING_LWIP
```

LwIP 协议栈的主线程 TCP 的参数（优先级，信箱大小，栈空间大小）也可以在 rtconfig.h 头文件中定义

```
/* tcp线程选项 */
#define RT_LWIP_TCPTHREAD_PRIORITY      120
#define RT_LWIP_TCPTHREAD_MBOX_SIZE     4
#define RT_LWIP_TCPTHREAD_STACKSIZE     1024
```

默认的 IP 地址，网关地址，子网掩码也可以在 rtconfig.h 头文件中定义（如果要使用 DHCP 方式分配，则需要定义RT_USING_DHCP宏）

```
/* 目标板IP地址 */
#define RT_LWIP_IPADDR0     192
#define RT_LWIP_IPADDR1     168
#define RT_LWIP_IPADDR2     1
#define RT_LWIP_IPADDR3     30

/* 网关地址 */
#define RT_LWIP_GWADDR0     192
#define RT_LWIP_GWADDR1     168
#define RT_LWIP_GWADDR2     1
#define RT_LWIP_GWADDR3     1

/* 子网掩码 */
#define RT_LWIP_MSKADDR0    255
#define RT_LWIP_MSKADDR1    255
```

```
#define RT_LWIP_MSKADDR2    255
#define RT_LWIP_MSKADDR3    0
```

14.2 缓冲区函数

14.2.1 netbuf_new() 原型声明

```
struct netbuf *netbuf_new(void)
```

分配一个 netbuf 结构，该函数并不会分配实际的缓冲区空间，只创建顶层的结构。netbuf 用完后，必须使用 netbuf_delete()回收。

14.2.2 netbuf_delete() 原型声明

```
void netbuf_delete(struct netbuf*)
```

回收先前通过调用 netbuf_new()函数创建的 netbuf 结构，任何通过 netbuf_alloc()函数分配给 netbuf 的缓冲区内存同样也会被回收。

例子：这个例子显示了使用 netbufs 的基本代码结构

```
void lw_thread(void *parameter)
{
    struct netbuf *buf;
    buf = netbuf_new();      /* 建立一个新的netbuf */
    netbuf_alloc(buf, 100); /* 为这个buf分配100 bytes */

    /* 对netbuf数据做一些处理 */
    /* [...] */

    netbuf_delete(buf);      /* 删除buf */
}
```

14.2.3 netbuf_alloc()原型声明

```
void *netbuf_alloc(struct netbuf *buf, int size)
```

为 netbuf buf 分配指定字节（bytes）大小的缓冲区内存。这个函数返回一个指针指向已分配的内存，任何先前已分配给 netbuf buf 的内存会被回收。刚分配的内存可以在以后使用 netbuf_free()函数回收。因为协议头应该要先于数据被发送，所以这个函数即为协议头也为实际的数据分配内存。

14.2.4 netbuf_free() 原型声明

```
int netbuf_free(struct netbuf *buf)
```

回收与 netbuf buf 相关联的缓冲区。如果还没有为 netbuf 分配缓冲区，这个函数不做任何事情。

14.2.5 netbuf_ref() 原型声明

```
int netbuf_ref(struct netbuf *buf, void *data, int size)
```

使数据指针指向的外部存储区与 netbuf buf 关联起来。外部存储区大小由 size 参数给出。任何先前已分配给 netbuf 的存储区会被回收。使用 netbuf_alloc()函数为 netbuf 分配存储区与先分配存储区——比如使用 malloc()函数——然后再使用 netbuf_ref()函数引用这块存储区相比，不同的是前者还要为协议头分配空间这样会使处理和发送缓冲区速度更快。

下面这个例子显示了 netbuf_ref()函数的简单用法

```
void lw_thread(void *parameter)
{
    struct netbuf *buf;
    char string[] = "A string";

    buf = netbuf_new();
    netbuf_ref(buf, string, sizeof(string));    /* 引用这个字符串 */

    /* [...] */

    netbuf_delete(buf);
}
```

14.2.6 netbuf_len() 原型声明

```
int netbuf_len(struct netbuf *buf)
```

返回 netbuf buf 中的数据长度，即使 netbuf 被分割为数据片断。对数据片断状的 netbuf 来说，通过调用这个函数取得的长度值并不等于 netbuf 中的第一个数据片断的长度，而是所有分片的长度。

14.2.7 netbuf_data() 原型声明

```
int netbuf_data(struct netbuf *buf, void **data, int *len)
```

这个函数用于获取一个指向 netbuf buf 中的数据的指针，同时还取得数据块的长度。参数 data 和 len 为结果参数，参数 data 用于接收指向数据的指针值，len 指针接收数据块长度。如果 netbuf 中的数据被分割为片断，则函数给出的指针指向 netbuf 中的第一个数据片断。应用程序必须使用片断处理函数 netbuf_first()和 netbuf_next()来取得 netbuf 中的完整数据。

关于如何使用 netbuf_data()函数请参阅下面 netbuf_next()函数说明中给出的例子。

14.2.8 netbuf_next() 原型声明

```
int netbuf_next(struct netbuf *buf)
```

函数修改netbuf中数据片断的指针以便指向netbuf中的下一个数据片断。返回值为0表明 netbuf中还有数据片断存在，大于0表明指针现在正指向最后一个数据片断，小于0表明已经到了最后一个数据片断的后面的位置，netbuf中已经没有数据片断了。

例子：这个例子显示了如何使用 netbuf_next()函数。我们假定这是一个函数的中间部分，并且其中的 buf 就是一个 netbuf 类型的变量。

```
void lw_thread(void* parameter)
{
    /* [...] */
    do
    {
        char *data;
        int len;

        netbuf_data(buf, &data, &len);  /* 获取一个指针指向数据片段中的数据 */

        do_something(data, len);        /* 对这些数据进行一些处理 */
    } while(netbuf_next(buf) >= 0);
    /* [...] */
}
```

14.2.9 netbuf_first() 原型声明

```
int netbuf_first(struct netbuf *buf)
```

复位netbuf buf中的数据片断指针，使其指向netbuf中的第一个数据片断。

14.2.10 netbuf_copy() 原型声明

```
void netbuf_copy(struct netbuf *buf, void *data, int len)
```

将netbuf buf中的所有数据复制到data指针指向的存储区，即使netbuf buf中的数据被分割为片断。len参数指定要复制数据的最大值。

下面的例子显示了 netbuf_copy()函数的简单用法。这里，协议栈分配 200 个字节的存储区用以保存数据，即使 netbuf buf 中的数据大于 200 个字节，也只会复制 200 个字节的数据。

```
void example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);

    /* 对这些数据进行一些处理 */
}
```

14.2.11 netbuf_chain()原型声明

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail)
```

将两个netbufs的首尾链接在一起，以使首部netbuf的最后一个数据片断成为尾部netbuf的第一个数据片断。函数被调用后，尾部netbuf会被回收，不能再使用。

14.2.12 netbuf_fromaddr() 原型声明

```
struct ip_addr *netbuf_fromaddr(struct netbuf *buf)
```

返回接收到的netbuf buf的主机IP地址。如果指定的netbuf还没有从网络收到，函数返回一个未定义值。netbuf_fromport()函数用于取得远程主机的端口号。

14.2.13 netbuf_fromport() 原型声明

```
unsigned short netbuf_fromport(struct netbuf *buf)
```

返回接收到的netbuf buf的主机端口号。如果指定的netbuf还没有从网络收到，函数返回一个不确定值。netbuf_fromaddr()函数用于取得远程主机的IP地址。

14.3 网络连接函数

14.3.1 netconn_new() 原型声明

```
struct netconn *netconn_new(enum netconn_type type)
```

建立一个新的连接数据结构，根据是要建立TCP还是UDP连接来选择参数值是NETCONN_TCP还是NETCONN_UCP。调用这个函数并不会建立连接并且没有数据被发送到网络中。

14.3.2 netconn_delete() 原型声明

```
void netconn_delete(struct netconn *conn)
```

删除连接数据结构conn，如果连接已经打开，调用这个函数将会关闭这个连接。

14.3.3 netconn_type() 原型声明

```
enum netconn_type netconn_type(struct netconn *conn)
```

返回指定的连接conn的连接类型。返回的类型值就是前面netconn_new()函数说明中提到的 NETCONN_TCP或者NETCONN_UDP。

14.3.4 netconn_peer() 原型声明

```
int netconn_peer(struct netconn *conn, struct ip_addr *addr, unsigned short *port)
```

这个函数用于获取连接的远程终端的IP地址和端口号。addr和port为结果参数，它们的值由函数设置。如果指定的连接conn并没有连接任何远程主机，则获得的结果值并不确定。

14.3.5 netconn_addr() 原型声明

```
int netconn_addr(struct netconn *conn, struct ip_addr **addr, unsigned short *port)
```

这个函数用于获取由conn指定的连接的本地IP地址和端口号。

14.3.6 netconn_bind() 原型声明

```
int netconn_bind(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

为参数conn指定的连接绑定本地IP地址和TCP或UDP端口号。如果addr参数为NULL则本地IP 地址由网络系统确定。

14.3.7 netconn_connect() 原型声明

```
int netconn_connect(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

对UDP连接，该函数通过addr和port参数设定发送的UDP消息要到达的远程主机的IP地址和端口号。对TCP，netconn_connect()函数打开与指定远程主机的连接。

14.3.8 netconn_listen() 原型声明

```
int netconn_listen(struct netconn *conn)
```

使参数conn指定的连接进入TCP监听（TCP LISTEN）状态。

14.3.9 netconn_accept() 原型声明

```
struct netconn *netconn_accept(struct netconn *conn)
```

阻塞进程直至从远程主机发出的连接请求到达参数conn指定的连接。这个连接必须处于监听（LISTEN）状态，因此在调用netconn_accept()函数之前必须调用netconn_listen()函数。与远程主机的连接建立后，函数返回新连接的结构。

例子：这个例子显示了如何在 2000 端口上打开一个 TCP 服务器。

```
void lw_thread(void* parameter)
{
    struct netconn *conn, *newconn;

    /*  建立一个连接结构 */
    conn = netconn_new(NETCONN_TCP);

    /*  将连接绑定到一个本地任意IP地址的2000端口上 */
    netconn_bind(conn, NULL, 2000);

    /*  告诉这个连接监听进入的连接请求 */
    netconn_listen(conn);

    /*  阻塞直至得到一个进入的连接 */
    newconn = netconn_accept(conn);

    /*  处理这个连接 */
    process_connection(newconn);

    /*  删除连接 */
    netconn_delete(newconn);
    netconn_delete(conn);
}
```

14.3.10 netconn_recv() 原型声明

```
struct netbuf *netconn_recv(struct netconn *conn)
```

阻塞进程，等待数据到达参数conn指定的连接。如果连接已经被远程主机关闭，则返回NULL，其它情况，函数返回一个包含着接收到的数据的netbuf。

例子：这是一个小例子，显示了 netconn_recv()函数的假定用法。我们假定在调用这个例子函数example_function()之前连接已经建立。

```
void example_function(struct netconn *conn)
{
    struct netbuf *buf;

    /*  接收数据直到其它主机关闭连接 */
    while((buf = netconn_recv(conn)) != NULL)
    {
                /* 对这些数据进行一些处理 */
        do_something(buf);
    }

    /*  连接现在已经被其它终端关闭，因此也关闭我们自己的连接 */
    netconn_close(conn);
}
```

14.3.11 netconn_write() 原型声明

```
int netconn_write(struct netconn *conn, void *data, int len, unsigned int flags)
```

这个函数只用于TCP连接。它把data指针指向的数据放在属于conn连接的输出队列。Len参数指定数据的长度，这里对数据长度没有任何限制。这个函数不需要应用程序明确的分配缓冲区（buffers)，因为这由协议栈来负责。flags参数有两种可能的状态，如下所示：

```
#define NETCONN_NOCOPY  0x00
#define NETCONN_COPY    0x01
```

当flags值为NETCONN_COPY时，data指针指向的数据被复制到为这些数据分配的内部缓冲区。这就允许这些数据在函数调用后可以直接修改，但是这会在执行时间和内存使用率方面降低效率。如果flags值为NETCONN_NOCOPY，数据不会被复制而是直接使用data指针来引用。这些数据在函数调用后不能被修改，因为这些数据可能会被放在当前指定连接的重发队列，并且会在里面逗留一段不确定的时间。当要发送的数据在ROM中因而数据不可变时这很有用。如果需要更多的控制数据的修改，则可以联合使用复制和不复制数据，如下面的例子所示。

这个例子显示了 netconn_write() 函数的基本用法。这里假定程序里的 data 变量在后面编辑修改，因此它被复制到内部缓冲区，方法是前文所讲的在调用 netconn_write() 函数时将 flags 参数值设为 NETCONN_COPY。text 变量包含了一个不能被编辑修改的字符串，因此它采用指针引用的方式以代替复制。

```
void lw_thread(void* parameter)
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /*  设置连接conn */
    /* [...] */
    /*  构造一些测试数据 */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);

    /*  这些数据可以被修改 */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;

    /*  关闭连接 */
    netconn_close(conn);
}
```

14.3.12 netconn_send() 原型声明

```
int netconn_send(struct netconn *conn, struct netbuf *buf)
```

使用参数conn指定的UDP连接发送参数buf中的数据。netbuf中的数据不能太大，因为没有使用IP分段。数据长度不能大于发送网络接口（outgoing network interface）的最大传输单元值（MTU）。因为目前还没有获取这个值的方法，这就需要采用其它的途径来避免超过MTU值，所以规定了一个上限，就是netbuf中包含的数据不能大于1000个字节。函数对要发送的数据大小没有进行校验，无论是非常小还是非常大，因而函数的执行结果是不确定的。

例子：这个例子显示了如何向 IP 地址为 10.0.0.1，UDP 端口号为 7000 的远程主机发送 UDP 数据。

```
void lw_thread(void* parameter)
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /* 建立一个新的连接 */
    conn = netconn_new(NETCONN_UDP);

    /* 设置远程主机的IP地址，执行这个操作后，addr.addr的值为0x0100000a */
    addr.addr = htonl(0x0a000001);

    /* 连接远程主机 */
    netconn_connect(conn, &addr, 7000);

    /* 建立一个新的netbuf    */
    buf = netbuf_new();
    data = netbuf_alloc(buf, 10);

    /* 构造一些测试数据 */
    for(i = 0; i < 10; i++)
        data[i]= i;

    /* 发送构造的数据 */
    netconn_send(conn, buf);

    /* 引用这个文本给netbuf */
    netbuf_ref(buf, text, sizeof(text));

    /* 发送文本 */
    netconn_send(conn, buf);

    /* 删除conn和buf */
    netconn_delete(conn);
    netconn_delete(buf);
}
```

14.3.13 netconn_close() 原型声明

```
int netconn_close(struct netconn *conn)
```

关闭参数conn指定的连接。

14.3.14 代码示例

这一节介绍了使用LwIP API编写的一个简单的web服务器，代码将在后面给出。这个简单的web服务器仅实现了HTTP/1.0协议的基本功能，显示了如何使用LwIP API实现一个实际地应用。

这个应用由单一线程组成，它负责接收来自网络的连接，响应HTTP请求，以及关闭连接。在这个应用中的线程lw_thread()负责必要的初始化及连接设置工作；连接设置过程是一个相当简单的例子，显示了如何使用最小限度API初始化连接。使用netconn_new()函数建立一个连接后，这个连接被绑定在TCP 80端口并且进入监听（LISTEN）状态，等待连接。一旦一个远程主机连接进来，netconn_accept()函数将返回连接的netconn结构。当这个连接已经被 process_connection() 函数处理后，必须使用 netconn_delete() 函数删除这个netconn。

在process_connection()函数，调用netconn_recv()函数接收一个netbuf，然后通过netbuf_data()函数获取一个指向实际的请求数据指针。这个指针指向netbuf中的第一个数据片断，并且我们希望它包含这个请求。这并不是一个不合实际的想法，因为我们只读取这个请求的前七个字节。如果我们想读取更多的数据，简单的方法是使用netbuf_copy()函数复制这个请求到一个连续的内存区然后在那里处理它。

这个简单的web服务器只响应HTTP GET对文件“/”的请求，并且检测到请求就会发出响应。这里，我们既需要发送针对HTML数据的HTTP头，还要发送HTML数据，所以对 netconn_write()函数调用了两次。因为我们不需要修改HTTP头和HTML数据，所以我们将 netconn_write()函数的flags参数值设为NETCONN_NOCOPY以避免复制。

最后，连接被关闭并且process_connection()函数返回。连接结构也会在这个调用后被删除。

下面就是这个应用的C代码：

```
/* 使用最小限度API实现的一个简单的HTTP/1.0服务器 */

#include "api.h"

/*  这是实际的web页面数据。大部分的编译器会将这些数据放在ROM里 */
const static char indexdata[] = "<html> \
    <head><title>A test page</title></head> \
    <body> \
    This is a small test page. \
    </body> \
    </html>";

const static char http_html_hdr[] = "Content-type: text/html\r\n\r\n";

/*  这个函数处理进入的连接 */
static void process_connection(struct netconn *conn)
{
    struct netbuf *inbuf;
    char *rq;
```

```
    int len;

    /*  从这个连接读取数据到inbuf，我们假定在这个netbuf中包含完整的请求 */
    inbuf = netconn_recv(conn);

    /*  获取指向netbuf中第一个数据片断的指针，在这个数据片段里我们希望包含这个请求 */
    netbuf_data(inbuf, &rq, &len);

    /*  检查这个请求是不是HTTP "GET /\r\n"  */
    if( rq[0] == 'G' &&
        rq[1] == 'E' &&
        rq[2] == 'T' &&
        rq[3] == ' ' &&
        rq[4] == '/' &&
        rq[5] == '\r' &&
        rq[6] == '\n')
    {
        /*  发送头部数据 */
        netconn_write(conn, http_html_hdr, sizeof(http_html_hdr), NETCONN_NOCOPY);

        /*  发送实际的web页面 */
        netconn_write(conn, indexdata, sizeof(indexdata), NETCONN_NOCOPY);

        /*  关闭连接 */
        netconn_close(conn);
    }
}

/* 线程入口 */
void lw_thread(void* paramter)
{
    struct netconn *conn, *newconn;

    /*  建立一个新的TCP连接句柄 */
    conn = netconn_new(NETCONN_TCP);

    /*  将连接绑定在任意的本地IP地址的80端口上 */
    netconn_bind(conn, NULL, 80);

    /*  连接进入监听状态 */
    netconn_listen(conn);

    /*  循环处理 */
    while(1)
    {
        /*  接受新的连接请求 */
        newconn = netconn_accept(conn);

        /*  处理进入的连接 */
        process_connection(newconn);

        /*  删除连接句柄 */
        netconn_delete(newconn);
```

```
    }

    return 0;
}
```

14.4 BSD Socket库

BSD Socket是在Unix下进行网络通信编程的API接口之一，也是网络编程的事实标准。

LwIP提供了一个轻型BSD Socket API的实现，为大量已有的网络应用程序提供了兼容的接口。LwIP的socket接口实现都在函数名前加有lwip_前缀，同时在头文件中把它采用宏定义的方式定义成标准的BSD Socket API接口。

14.4.1 分配一个socket

```
int lwip_socket(int domain, int type, int protocol);
int socket(int domain, int type, int protocol);
```

应用程序在使用socket前，首先必须拥有一个socket，调用socket()函数可以为应用程序分配一个socket。socket()函数的参数用于指定所需要的socket的类型。

domain参数可以支持如下类型：

```
#define AF_UNSPEC       0
#define AF_INET         2
#define PF_INET         AF_INET
#define PF_UNSPEC       AF_UNSPEC
```

type参数可以支持如下类型：

```
/* Socket protocol types (TCP/UDP/RAW) */
#define SOCK_STREAM     1
#define SOCK_DGRAM      2
#define SOCK_RAW        3
```

这些定义的意义如下：

- SOCK_STREAM：流套接字。使用该套接字，进程可以使用 TCP 进行通信。流套接字提供没有记录边界的双向、可靠、有序且不重复的数据流。建立连接之后，可以将数据作为字节流从这些套接字中读取或向其中写入。
- SOCK_DGRAM：数据报套接字。使用该套接字，线程可以使用 UDP 进行通信。数据报套接字支持双向消息流。数据报套接字上的线程接收消息的顺序可能不同于发送消息的顺序。数据报套接字上的线程可能会接收重复消息。通过数据报套接字发送的消息可能会被丢弃。但数据中的记录边界会被保留。
- SOCK_RAW：原始套接字。该套接字提供对 ICMP 的访问。原始套接字并不适用于大多数应用。提供原始套接字是为了支持开发新的通信协议，或者为了访问现有协议的更加深奥的功能。

protocol参数可以支持如下类型：

```
#define IPPROTO_IP      0
#define IPPROTO_TCP     6
#define IPPROTO_UDP     17
#define IPPROTO_UDPLITE 136
```

14.4.2 绑定socket到地址

```
int lwip_bind(int s, struct sockaddr *name, socklen_t namelen);
int bind(int s, struct sockaddr *name, socklen_t namelen);
```

bind()调用为socket绑定一个本地地址。本地地址由name指定，其长度由namelen指定。sockaddr结构定义如下：

```
struct sockaddr {
  u8_t sa_len;
  u8_t sa_family;
  char sa_data[14];
};
```

14.4.3 建立到远端socket的连接

```
int lwip_connect(int s, const struct sockaddr *name, socklen_t namelen);
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

调用connect()函数连接socket到一个远端地址。调用时需要指定一个远端连接的地址。

14.4.4 接收一个连接

```
int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

accept()函数等待一个连接请求到达指定的TCP socket，而这个socket先前已经通过调用listen()函数进入监听状态。

对accept()函数的调用会根据socket选项情况阻塞线程直至与远程主机建立连接或非阻塞方式返回。这个addr参数是一个结果参数，它的值由accept()函数设置，这个值其实就是远程主机的地址。当新的连接已经建立，LwIP将把远程主机的IP地址和端口号保存到addr参数后，分配一个新的socket标识符，然后accept函数返回这个标识符。

14.4.5 监听连接

```
int lwip_listen(int s, int backlog);
int listen(int s, int backlog);
```

调用listen()函数相当于调用LwIP API函数netconn_listen()，且只能用于TCP连接。与BSD Socket API中listen函数不同的是，BSD Socket允许应用程序指定等待连接队列的大小（backlog参数指定）。LwIP协议栈只支持范围在0 - 255内值的backlog参数。

14.4.6 发送数据

```
int lwip_send(int s, const void *dataptr, int size, unsigned int flags);
int lwip_sendto(int s, const void *dataptr, int size, unsigned int flags,
    struct sockaddr *to, socklen_t tolen);
int lwip_write(int s, const void *dataptr, int size);

int send(int s, const void *dataptr, int size, unsigned int flags);
int sendto(int s, const void *dataptr, int size, unsigned int flags,
    struct sockaddr *to, socklen_t tolen);
int write(int s, const void *dataptr, int size);
```

send()调用用于在参数s指定的已连接的数据报或流socket上发送输出数据，其中参数s为已连接的socket描述符；dataptr指向存有发送数据的缓冲区的指针，其长度由 size 指定。flags支持的参数包括：

```
/* Flags we can use with send and recv. */
#define MSG_PEEK        0x01    /* 查看当前数据       */
#define MSG_WAITALL     0x02    /* 等到所有的信息到达时才返回，不支持 */
#define MSG_OOB         0x04    /* 带外数据，不支持   */
#define MSG_DONTWAIT    0x08    /* 非阻塞模式         */
#define MSG_MORE        0x10    /* 发送更多           */
```

sendto()调用用于将数据由指定的socket传给远方主机。参数to用来指定要传送到的网络地址，结构sockaddr请参考bind()。参数tolen为sockaddr的结果长度。

write()调用用于往参数s指定的已连接的socket中写入数据，其中参数s为已连接的本地socket描述符；dataptr指向存有发送数据的缓冲区的指针，其长度由 size 指定。

14.4.7 接收数据

recv()调用用于在参数s指定的已连接的数据报或流socket上接收输入数据。其中参数s为已连接的socket描述符；mem指向用于保存接收数据的的缓冲区指针，其能够存放的最大长度由 len 指定。flags支持的参数包括：

```
/* Flags we can use with send and recv. */
#define MSG_PEEK        0x01    /* 查看当前数据       */
#define MSG_WAITALL     0x02    /* 等到所有的信息到达时才返回，不支持 */
#define MSG_OOB         0x04    /* 带外数据，不支持   */
#define MSG_DONTWAIT    0x08    /* 非阻塞模式         */
#define MSG_MORE        0x10    /* 发送更多           */
```

recvfrom()调用用于在指定的socket上接收远方主机传递过来的数据。参数from用来指定欲传送的网络地址，结构sockaddr请参考bind()函数。参数fromlen为sockaddr的结构长度。

read()调用用于在参数s指定的已连接的socket中读取数据，其中参数s为已连接的本地socket描述符；mem指向用于保存接收数据的的缓冲区指针，其能够存放的最大长度由 len 指定。

14.4.8 输入/输出多路复用

```
int lwip_select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
                struct timeval *timeout);
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
                struct timeval *timeout);
```

select()调用用来检测一个或多个socket的状态。对每一个socket来说，这个调用可以请求读、写或错误状态方面的信息。请求给定状态的socket集合由一个fd_set结构指示。在返回时，此结构被更新，以反映那些满足特定条件的socket的子集，同时 select() 调用返回满足条件的socket的数目。

Note: select函数在RT-Thread中不能用于文件描述符操作。

14.4.9 关闭socket

```
int lwip_close(int s);
int close(int s);
```

close()关闭socket s，并释放分配给该socket的资源；如果s涉及一个打开的TCP连接，则该连接被释放。

14.4.10 UDP例子

下面的是一个如何在RT-Thread上使用BSD socket接口的UDP服务端例子，当把这个代码加入到RT-Thread时，它会自动向finsh 命令行添加一个udpserv命令，在finsh上执行udpserv()函数即可启动这个UDP服务端，它是在端口5000上进行监听。

当服务端接收到数据时，它将把数据打印到控制终端中；

如果服务端接收到exit字符串时，服务端将退出服务。

例子代码如下：

```
1  #include <rtthread.h>
2  #include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */
3
4  void udpserv(void* paramemter)
5  {
6     int sock;
7     int bytes_read;
8     char *recv_data;
9     rt_uint32_t addr_len;
10    struct sockaddr_in server_addr, client_addr;
11
12    /* 分配接收用的数据缓冲 */
13    recv_data = rt_malloc(1024);
14    if (recv_data == RT_NULL)
15    {
16        /* 分配内存失败，返回 */
17        rt_kprintf("No memory\n");
18        return;
19    }
```

```
20
21      /* 创建一个socket，类型是SOCK_DGRAM，UDP类型 */
22      if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
23      {
24          rt_kprintf("Socket error\n");
25
26          /* 释放接收用的数据缓冲 */
27          rt_free(recv_data);
28          return;
29      }
30
31      /* 初始化服务端地址 */
32      server_addr.sin_family = AF_INET;
33      server_addr.sin_port = htons(5000);
34      server_addr.sin_addr.s_addr = INADDR_ANY;
35      rt_memset(&(server_addr.sin_zero),0, sizeof(server_addr.sin_zero));
36
37      /* 绑定socket到服务端地址 */
38      if (bind(sock,(struct sockaddr *)&server_addr,
39               sizeof(struct sockaddr)) == -1)
40      {
41          /* 绑定地址失败 */
42          rt_kprintf("Bind error\n");
43
44          /* 释放接收用的数据缓冲 */
45          rt_free(recv_data);
46          return;
47      }
48
49      addr_len = sizeof(struct sockaddr);
50      rt_kprintf("UDPServer Waiting for client on port 5000...\n");
51
52      while (1)
53      {
54          /* 从sock中收取最大1024字节数据 */
55          bytes_read = recvfrom(sock, recv_data, 1024, 0,
56                                (struct sockaddr *)&client_addr, &addr_len);
57          /* UDP不同于TCP，它基本不会出现收取的数据失败的情况，除非设置了超时等待 */
58
59          recv_data[bytes_read] = '\0'; /* 把末端清零 */
60
61          /* 输出接收的数据 */
62          rt_kprintf("\n(%s , %d) said : ",inet_ntoa(client_addr.sin_addr),
63                     ntohs(client_addr.sin_port));
64          rt_kprintf("%s", recv_data);
65
66          /* 如果接收数据是exit，退出 */
67          if (strcmp(recv_data, "exit") == 0)
68          {
69              lwip_close(sock);
70
71              /* 释放接收用的数据缓冲 */
72              rt_free(recv_data);
```

```
                break;
            }
        }

        return;
    }

    #ifdef RT_USING_FINSH
    #include <finsh.h>
    /* 输出udpserv函数到finsh shell中 */
    FINSH_FUNCTION_EXPORT(udpserv, startup udp server);
    #endif
```

下面的是一个如何在RT-Thread上使用BSD socket接口的UDP客户端例子。当把这个代码加入到RT-Thread时，它会自动向finsh 命令行添加一个udpclient命令，在finsh上执行 udpclient(url, port) 函数即可启动这个UDP客户端，url指定了这个客户端连接到的服务端地址或域名，port是相应的端口号。

当UDP客户端启动后，它将连续发送5次“This is UDP Client from RT-Thread.”的字符串给服务端，然后退出。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

const char send_data[] = "This is UDP Client from RT-Thread.\n"; /* 发送用到的数据 */
void udpclient(const char* url, int port, int count)
{
    int sock;
    struct hostent *host;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host= (struct hostent *) gethostbyname(url);

    /* 创建一个socket，类型是SOCK_DGRAM，UDP类型 */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        return;
    }

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

    /* 总计发送count次数据 */
    while (count)
    {
```

```
        /* 发送数据到服务远端 */
        sendto(sock, send_data, strlen(send_data), 0,
               (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

        /* 线程休眠一段时间 */
        rt_thread_delay(50);

        /* 计数值减一 */
        count --;
    }

    /* 关闭这个socket */
    lwip_close(sock);
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出udpclient函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(udpclient, startup udp client);
#endif
```

14.4.11 TCP例子

下面的是一个如何在RT-Thread上使用BSD socket接口的TCP服务端例子，当把这个代码加入到RT-Thread时，它会自动向finsh 命令行添加一个tcpserv命令，在finsh上执行tcpserv()函数即可启动这个TCP服务端，它是在设备上端口5000进行监听。

当有TCP客户向这个服务端进行连接后，只要服务端接收到数据，它立即向客户端发送"This is TCP Server from RT-Thread."的字符串。

如果服务端接收到q或Q字符串时，服务器将主动关闭这个TCP连接。如果服务端接收到exit字符串时，服务端将退出服务。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD Socket接口必须包含sockets.h这个头文件 */

static const char send_data[] = "This is TCP Server from RT-Thread."; /* 发送用到的数据 */
void tcpserv(void* parameter)
{
    char *recv_data; /* 用于接收的指针，后面会做一次动态分配以请求可用内存 */
    rt_uint32_t sin_size;
    int sock, connected, bytes_received;
    struct sockaddr_in server_addr, client_addr;
    rt_bool_t stop = RT_FALSE; /* 停止标志 */

    recv_data = rt_malloc(1024); /* 分配接收用的数据缓冲 */
    if (recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
```

```
18      }
19
20      /* 一个socket在使用前，需要预先创建出来，指定SOCK_STREAM为TCP的socket */
21      if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22      {
23          /* 创建失败的错误处理 */
24          rt_kprintf("Socket error\n");
25
26          /* 释放已分配的接收缓冲 */
27          rt_free(recv_data);
28          return;
29      }
30
31      /* 初始化服务端地址 */
32      server_addr.sin_family = AF_INET;
33      server_addr.sin_port = htons(5000); /* 服务端工作的端口 */
34      server_addr.sin_addr.s_addr = INADDR_ANY;
35      rt_memset(&(server_addr.sin_zero),8, sizeof(server_addr.sin_zero));
36
37      /* 绑定socket到服务端地址 */
38      if (bind(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)
39      {
40          /* 绑定失败 */
41          rt_kprintf("Unable to bind\n");
42
43          /* 释放已分配的接收缓冲 */
44          rt_free(recv_data);
45          return;
46      }
47
48      /* 在socket上进行监听 */
49      if (listen(sock, 5) == -1)
50      {
51          rt_kprintf("Listen error\n");
52
53          /* release recv buffer */
54          rt_free(recv_data);
55          return;
56      }
57
58      rt_kprintf("\nTCPServer Waiting for client on port 5000...\n");
59      while(stop != RT_TRUE)
60      {
61          sin_size = sizeof(struct sockaddr_in);
62
63          /* 接受一个客户端连接socket的请求，这个函数调用是阻塞式的 */
64          connected = accept(sock, (struct sockaddr *)&client_addr, &sin_size);
65          /* 返回的是连接成功的socket */
66
67          /* 接受返回的client_addr指向了客户端的地址信息 */
68          rt_kprintf("I got a connection from (%s , %d)\n",
69                     inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));
70
```

```
71          /* 客户端连接的处理 */
72          while (1)
73          {
74              /* 发送数据到connected socket */
75              send(connected, send_data, strlen(send_data), 0);
76
77              /* 从connected socket中接收数据，接收buffer是1024大小，但并不一定能够收到1024大小的数据 */
78              bytes_received = recv(connected,recv_data, 1024, 0);
79              if (bytes_received < 0)
80              {
81                  /* 接收失败，关闭这个connected socket */
82                  lwip_close(connected);
83                  break;
84              }
85
86              /* 有接收到数据，把末端清零 */
87              recv_data[bytes_received] = '\0';
88              if (strcmp(recv_data , "q") == 0 || strcmp(recv_data , "Q") == 0)
89              {
90                  /* 如果是首字母是q或Q，关闭这个连接 */
91                  lwip_close(connected);
92                  break;
93              }
94              else if (strcmp(recv_data, "exit") == 0)
95              {
96                  /* 如果接收的是exit，则关闭整个服务端 */
97                  lwip_close(connected);
98                  stop = RT_TRUE;
99                  break;
100             }
101             else
102             {
103                 /* 在控制终端显示收到的数据 */
104                 rt_kprintf("RECIEVED DATA = %s \n" , recv_data);
105             }
106         }
107     }
108
109     /* 退出服务 */
110     lwip_close(sock);
111
112     /* 释放接收缓冲 */
113     rt_free(recv_data);
114
115     return ;
116 }
117
118 #ifdef RT_USING_FINSH
119 #include <finsh.h>
120 /* 输出tcpserv函数到finsh shell中 */
121 FINSH_FUNCTION_EXPORT(tcpserv, startup tcp server);
122 #endif
```

下面的是一个如何在RT-Thread上使用BSD socket接口的TCP客户端例子。当把这个代码加入到RT-Thread时，它会自动向finsh 命令行添加一个tcpclient命令，在finsh上执行tcpclient(url, port)函数即可启动这个TCP服务端，url指定了这个客户端连接到的服务端地址或域名，port是相应的端口号。

当TCP客户端连接成功时，它会接收服务端传过来的数据。当有数据接收到时，如果是以q或Q开头，它将主动断开这个连接；否则会把接收的数据在控制终端中打印出来，然后发送“This is TCP Client from RT-Thread.”的字符串。

例子代码如下：

```c
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

static const char send_data[] = "This is TCP Client from RT-Thread."; /* 发送用到的数据 */
void tcpclient(const char* url, int port)
{
    char *recv_data;
    struct hostent *host;
    int sock, bytes_received;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host = gethostbyname(url);

    /* 分配用于存放接收数据的缓冲 */
    recv_data = rt_malloc(1024);
    if (recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
    }

    /* 创建一个socket，类型是SOCKET_STREAM，TCP类型 */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        /* 创建socket失败 */
        rt_kprintf("Socket error\n");

        /* 释放接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

    /* 连接到服务端 */
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)
    {
```

```
44          /* 连接失败 */
45          rt_kprintf("Connect error\n");
46
47          /*释放接收缓冲 */
48          rt_free(recv_data);
49          return;
50      }
51
52      while(1)
53      {
54          /* 从sock连接中接收最大1024字节数据 */
55          bytes_received = recv(sock, recv_data, 1024, 0);
56          if (bytes_received < 0)
57          {
58              /* 接收失败，关闭这个连接 */
59              lwip_close(sock);
60
61              /* 释放接收缓冲 */
62              rt_free(recv_data);
63              break;
64          }
65
66          /* 有接收到数据，把末端清零 */
67          recv_data[bytes_received] = '\0';
68
69          if (strcmp(recv_data , "q") == 0 || strcmp(recv_data , "Q") == 0)
70          {
71              /* 如果是首字母是q或Q，关闭这个连接 */
72              lwip_close(sock);
73
74              /* 释放接收缓冲 */
75              rt_free(recv_data);
76              break;
77          }
78          else
79          {
80              /* 在控制终端显示收到的数据 */
81              rt_kprintf("\nRecieved data = %s " , recv_data);
82          }
83
84          /* 发送数据到sock连接 */
85          send(sock,send_data,strlen(send_data), 0);
86      }
87
88      return;
89  }
90
91  #ifdef RT_USING_FINSH
92  #include <finsh.h>
93  /* 输出tcpclient函数到finsh shell中 */
94  FINSH_FUNCTION_EXPORT(tcpclient, startup tcp client);
95  #endif
```

CHAPTER

FIFTEEN

图形用户界面

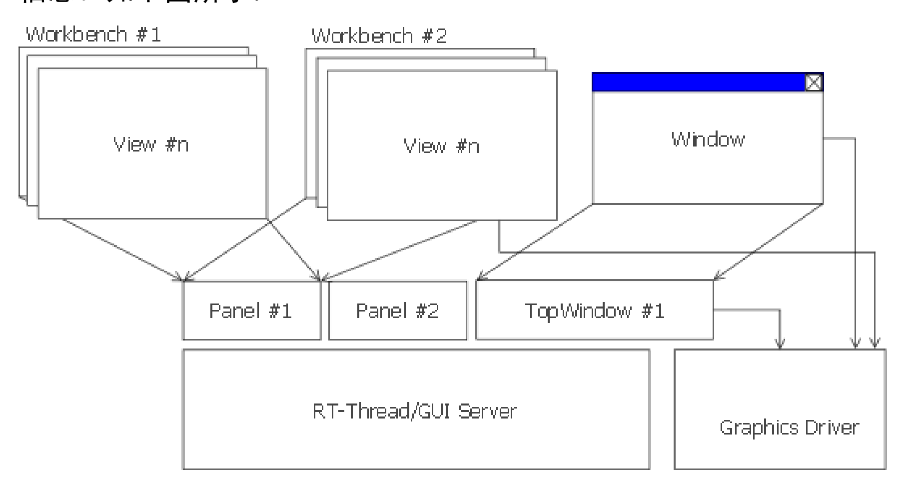
15.1 简介

RT-Thread/GUI是一个图形用户界面(Graphic User Interface)，它专为RT-Thread操作系统而开发，并在一些地方采用了RT-Thread特有功能以和RT-Thread无缝的整合起来。这个图形用户界面组件能够为RT-Thread上的应用程序提供人机界面交互的功能，例如人机界面设备，设备信息显示，播放器界面等。图形用户界面组件的功能包括：

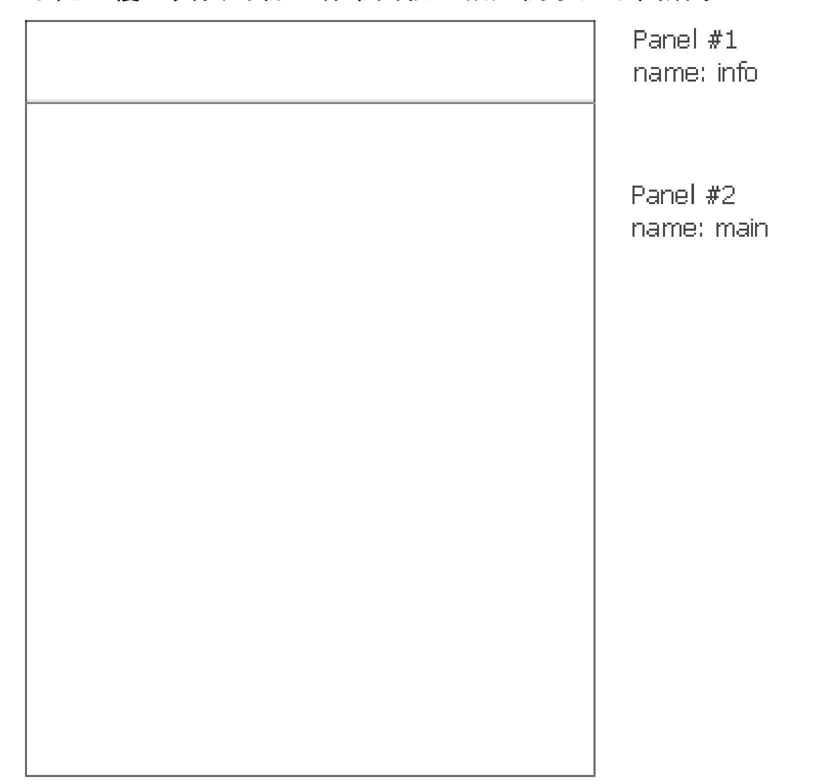
- 多线程图形用户界面；
- 依赖于RT-Thread线程调度器的实时图形用户界面；
- 创新式的在嵌入式系统中引入面板的概念(panel)，缩小了多线程，多窗口图形用户界面编程代价：
  - workbench，对应多线程；
  - view，对应不同的用户界面视图；
  - window，对应多窗口；
- C语言方式的全面向对象设计：
  - 对象具备运行时类型信息；
  - 对象自动销毁，使得内存的管理更为轻松；
- 界面主题支持；
- 中文文本显示支持；
- 丰富的控件支持：
  - button，checkbox，radiobox
  - textbox
  - progressbar，slider
  - 列表视图，文件列表视图
  - 等等

15.2 构架

RT-Thread/GUI采用传统的客户端/服务端(C/S)的结构，但和传统的客户端/服务端构架，把绘画操作放于服务端不同的是，绘画操作完全有客户端自行完成。服务端仅维护着各个客户端的位置信息。如下图所示：

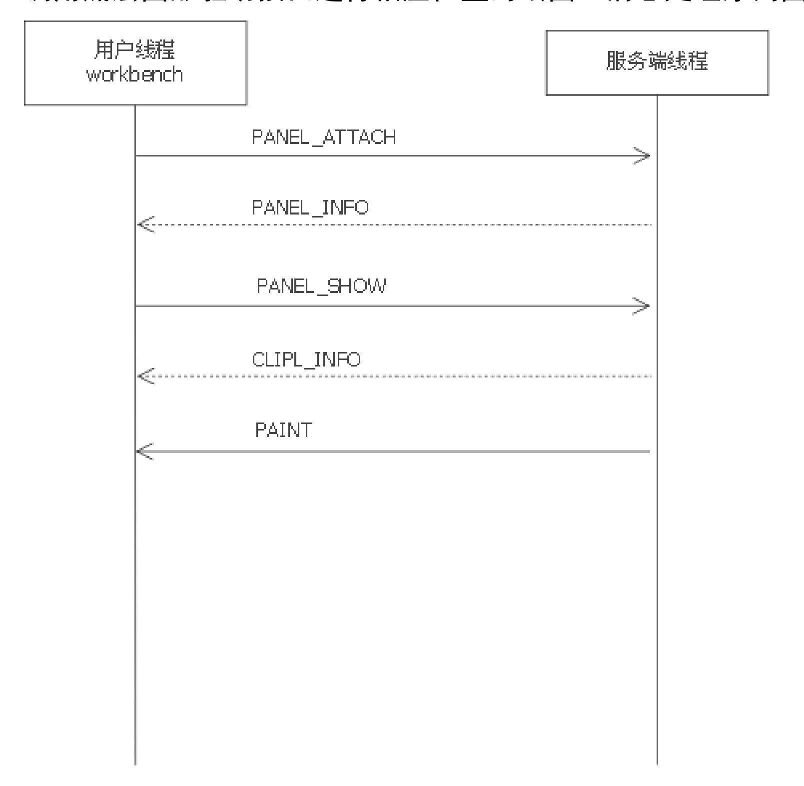


RT-Thread/GUI Server图形用户界面的服务端：当一个图形应用(workbench)或窗口需要创建时，需要向GUI服务端请求允许创建相应的实例。GUI服务端将记录下相应的位置信息（或者说RTGUI服务端管理着屏幕的可视区域信息）。在RT-Thread/GUI中，屏幕会被分隔成数个互不相重叠的面板(panel)，当然面板的数目也能只是一个（即全屏），或所有面板拼接成的总体面积并不完全覆盖真实的物理屏幕面积。相应例子如下图所示：



面板1(panel #1)是一个系统信息面板，用于显示系统的一些信息，例如日期时间，电池容量等。面板2(panel #2)是一个主面板，用于主要应用程序的显示交互。

workbench是附着在面板上的应用程序，每一个workbench具备独立的线程执行上下文环境，相应线程具备自己的消息邮箱，其事件处理循环即为处理达到的消息事件。创建一个workbench时，应用程序需要主动发送相应事件给GUI服务端并且给出它附着(attach)的面板名称。如果GUI服务端获得workbench的附着请求后，查询系统中确实存在相应面板，它将把相应面板的信息返回给workbench以允许它在上面进行显示。当workbench获得了面板信息后，它将能够在绘图时直接调用底层图形驱动接口进行相应位置的绘图。消息处理序列图如下图所示：



窗口(window)在RT-Thread/GUI中被设计成始终位于panel上方的顶层显示，更多适合于一些提示性消息显示等。

15.3 文件目录

RT-Thread/GUI位于RT-Thread源代码工程中的rtgui目录，在这个目录下分别包含了四个子目录：

目录名	作用
common	GUI的一些公共实现，例如DC，Image，字体，对象系统，默认主题等。
include	GUI的头文件，如应用程序使用RT-Thread/GUI，需要在添加这个路面到头文件包含路径中。
server	服务端实现。
widgets	控件实现，例如workbench，view，button，textbox等。

15.4 服务端

RT-Thread/GUI服务端包含一个事件处理线程rtgui，主要分成几个模块：

- GUI Server，服务端事件处理主线程。

- Panel，面板及面板之上的线程维护。
- TopWin，顶层窗口信息维护。
- Mouse，鼠标相关信息处理。

15.4.1 面板

RT-Thread/GUI服务端维护着屏幕的面板(Panel)划分，并且面板与面板之间是相互不重叠。所以在使用RT-Thread/GUI之前请正确的注册相应面板信息。注册面板可以通过如下函数接口进行：

```
void rtgui_panel_register(char* name, rtgui_rect_t* extent);
```

参数name，指示出要创建的面板名称。由于workbench在创建时需要按照面板名称的方式附着，所以面板名称不能命名成已经存在的面板名称；参数extent，指示出要创建面板的位置信息。

如上面的面板，分别包含了两个面板： panel #1：info面板。 panel #2：main面板。

可以采用下面的代码进行注册：

```
1  #include <rtgui/rtgui.h>
2  #include <rtgui/rtgui_server.h>
3
4  /*
5   * 应用于240x320上的双面板注册
6   * Info面板: (0,  0) - (240,  25)
7   * 主面板  : (0, 25) - (240, 320)
8   */
9  void panel_init(void)
```

```
{
    rtgui_rect_t rect;

    /* 注册Info面板位置信息 */
    rect.x1 = 0;
    rect.y1 = 0;
    rect.x2 = 240;
    rect.y2 = 25;
    rtgui_panel_register("info", &rect);

    /* 注册主面板位置信息 */
    rect.x1 = 0;
    rect.y1 = 25;
    rect.x2 = 320;
    rect.y2 = 320;
    rtgui_panel_register("main", &rect);
    rtgui_panel_set_default_focused("main");
}
```

在上面这个例子中，为240x320的屏幕注册了两个面板，一个面板是Info面板，位置信息是(0,0) - (240, 25)。一个面板是主面板，位置信息是(0, 25) - (240, 320)。

在整个系统中，也能够仅仅只注册一个面板（即全屏），相应代码如下：

```
#include <rtgui/rtgui.h>
#include <rtgui/rtgui_server.h>

/*
 * 应用于240x320上的单面板注册
 */
void panel_init(void)
{
    rtgui_rect_t rect;

    /* 注册主面板位置信息 */
    rect.x1 = 0;
    rect.y1 = 0;
    rect.x2 = 240;
    rect.y2 = 320;
    rtgui_panel_register("main", &rect);
    rtgui_panel_set_default_focused("main");
}
```

在上面这个例子中，为240x320的屏幕注册了一个单面板，并且包括了整体屏幕。

和面板相关的事件包括：

- RTGUI_EVENT_PANEL_ATTACH, 这个事件由应用发送给服务端线程，以附着到面板上。
- RTGUI_EVENT_PANEL_DETACH, 应用线程从面板上脱离
- RTGUI_EVENT_PANEL_SHOW, 应用线程请求显示于附着的面板上
- RTGUI_EVENT_PANEL_HIDE, 隐藏当前面板上的线程
- RTGUI_EVENT_PANEL_INFO, 当应用线程请求附着面板时，服务端返回相应的面板信息

- RTGUI_EVENT_PANEL_FULLSCREEN, 应用线程请求面板进行全屏显示(当前版本不支持)
- RTGUI_EVENT_PANEL_NORMAL, 应用线程请求面板恢复到原始状态(当前版本不支持)

15.4.2 窗口

当应用线程需要创建窗口时，亦需要向GUI服务端请求创建，并把它的位置信息报给服务端。服务端在收到窗口创建请求后，需要把它的位置信息加入到topwin列表中，并根据它的情况决定是否添加边框、标题信息。

注：窗口的边框和标题栏由GUI服务端维护，并进行显示。

和窗口相关的事件包括：

- RTGUI_EVENT_WIN_CREATE, 创建窗口事件
- RTGUI_EVENT_WIN_DESTROY, 删除窗口事件
- RTGUI_EVENT_WIN_SHOW, 显示窗口事件
- RTGUI_EVENT_WIN_HIDE, 隐藏窗口事件
- RTGUI_EVENT_WIN_ACTIVATE, 激活窗口事件
- RTGUI_EVENT_WIN_DEACTIVATE, 去激活窗口事件
- RTGUI_EVENT_WIN_CLOSE, 关闭窗口事件
- RTGUI_EVENT_WIN_MOVE, 移动窗口事件
- RTGUI_EVENT_WIN_RESIZE, 更改窗口的大小信息(当前版本不支持)

15.4.3 鼠标与键盘

鼠标与键盘的处理亦由GUI服务端处理，它们都转换成消息事件的方式进行处理。所以鼠标、键盘驱动最主要的方法就是：把相应的鼠标状态和键值转换成事件的形式发送给GUI服务端线程。发送给GUI服务端的函数是：

```
void rtgui_server_post_event(struct rtgui_event* event, rt_size_t size);
```

event，指定发送的消息事件。size，消息事件的大小。

相应的事件类型如下：

- RTGUI_EVENT_MOUSE_MOTION, 鼠标移到事件
- RTGUI_EVENT_MOUSE_BUTTON, 鼠标按键事件
- RTGUI_EVENT_KBD, 键盘按键事件

相应的使用例子如下：

```
struct rtgui_event_kbd kbd_event;

/* 初始化键盘按键事件 */
RTGUI_EVENT_KBD_INIT(&kbd_event);
kbd_event.mod   = RTGUI_KMOD_NONE;
kbd_event.unicode = 0;
```

```
kbd_event.key   = RTGUIK_HOME;
kbd_event.type = RTGUI_KEYDOWN;

/* 发送键盘事件到服务端 */
rtgui_server_post_event(&(kbd_event.parent), sizeof(kbd_event));
```

15.5 客户端

RT-Thread/GUI是为多线程而设计的，其中客户端会负责大部分UI操作，服务端提供的功能主要是客户端(线程)的管理及鼠标、键盘事件的派发。

一个客户端是一个独立的可执行体：线程。脱离了线程的存在将无法进行相应的UI操作，客户端线程的主要操作可以分成两大类：

- 事件处理；
- 绘图操作。

这两类操作都必须在客户端线程执行环境下进行，即：

- 不可在中断服务例程中进行；
- 不可在非相应客户端线程执行环境下执行上面两类操作。

客户端根据上层应用的不同相应的分成了两类：

- workbench
- window

但是其中有一个例外，window可以是一个独立窗口(具备独立线程)，也可以是workbench下的附属窗口(事件处理，绘图操作依附于workbench线程)。

要成为一个GUI客户端的前提条件是建立相应的事件处理消息队列（以接收服务端发送过来的消息），可以使用如下流程完成：

```
/*
 * 程序清单：建立一个workbench应用
 */

/* workbench线程的入口函数声明 */
extern static void workbench_entry(void* parameter);

/* UI应用程序的初始化 */
void ui_application_init()
{
    rt_thread_t tid;

    /* 创建一个线程用于workbench应用 */
    tid = rt_thread_create("wb",
        workbench_entry, RT_NULL,
        2048, 25, 10);

    /* 启动线程 */
    if (tid != RT_NULL) rt_thread_startup(tid);
```

```
20  }
21
22  /* workbench应用入口 */
23  static void workbench_entry(void* parameter)
24  {
25      rt_mq_t mq;
26      struct rtgui_view* view;
27      struct rtgui_workbench* workbench;
28
29      /* 创建相应的事件处理消息队列 */
30  #ifdef RTGUI_USING_SMALL_SIZE
31      mq = rt_mq_create("workbench", 32, 8, RT_IPC_FLAG_FIFO);
32  #else
33      mq = rt_mq_create("workbench", 256, 8, RT_IPC_FLAG_FIFO);
34  #endif
35      /* 注册成为GUI线程 */
36      rtgui_thread_register(rt_thread_self(), mq);
37
38      /* 创建workbench */
39      workbench = rtgui_workbench_create("main", "workbench");
40      if (workbench == RT_NULL) return;
41
42      /* 在workbench创建成功后，可以加入view或window等，此处略 */
43
44      /* 执行workbench的事件循环 */
45      rtgui_workbench_event_loop(workbench);
46
47      /* 当从事件循环中退出时，一般代表这个workbench已经关闭 */
48
49      /* 去注册GUI线程 */
50      rtgui_thread_deregister(rt_thread_self());
51      /* 删除相应的消息队列 */
52      rt_mq_delete(mq);
53  }
```

从上面的代码例程可以看到，创建一个workbench应用，最主要的有三点：

- 创建相应的线程(执行环境)；
- 提供相应的事件处理消息队列；
- 执行workbench的事件循环。

注：目前RT-Thread/GUI支持两种模式，小模式和标准模式。当使用小模式时，相应的内存占用更少。而标准模式则支持更多的回调函数，并且支持自动布局引擎。当使用小模式时，消息队列的单个消息可以相应的小一些，但不能小于32字节。

下面的代码例程是创建一个独立window应用的例子：

```
1  /*
2   * 程序清单：建立一个独立window
3   */
4
5  /* window线程的入口函数声明 */
6  extern static void window_entry(void* parameter);
```

```

/* UI应用程序的初始化 */
void ui_application_init()
{
    rt_thread_t tid;

    /* 创建一个线程用于workbench应用 */
    tid = rt_thread_create("win",
        window_entry, RT_NULL,
        2048, 25, 10);

    /* 启动线程 */
    if (tid != RT_NULL) rt_thread_startup(tid);
}

/* workbench应用入口 */
static void window_entry(void* parameter)
{
    rt_mq_t mq;
    struct rtgui_view* view;
    struct rtgui_win* win;
    rtgui_rect_t rect = {0, 0, 200, 120};

    /* 创建相应的事件处理消息队列 */
#ifdef RTGUI_USING_SMALL_SIZE
    mq = rt_mq_create("win", 32, 8, RT_IPC_FLAG_FIFO);
#else
    mq = rt_mq_create("win", 256, 8, RT_IPC_FLAG_FIFO);
#endif
    /* 注册成为GUI线程 */
    rtgui_thread_register(rt_thread_self(), mq);

    /* 创建win, parent为空即代表它是一个独立窗口 */
    win = rtgui_win_create(RT_NULL, "win", &rect, RTGUI_WIN_STYLE_DEFAULT);
    if (win == RT_NULL) return;

    /* 显示窗口 */
    rtgui_win_show(win, RT_FALSE);

    /* 执行window的事件循环 */
    rtgui_win_event_loop(win);

    /* 当从事件循环中退出时，一般代表这个win已经关闭 */

    /* 去注册GUI线程 */
    rtgui_thread_deregister(rt_thread_self());
    /* 删除相应的消息队列 */
    rt_mq_delete(mq);
}
```

同样，上面的创建一个独立窗口的代码例程主要的依然是三点：

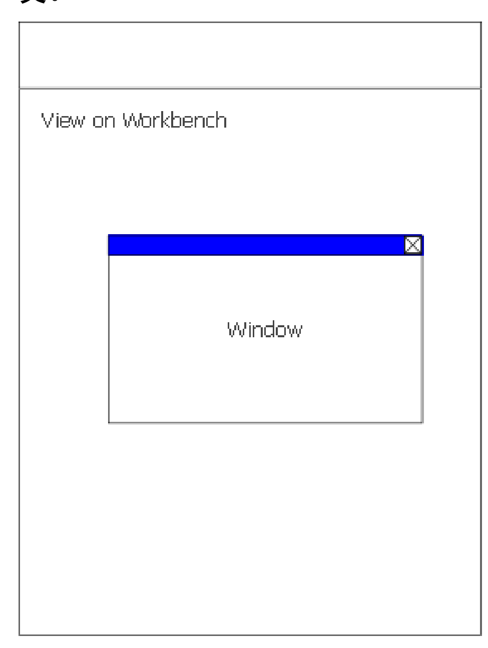
- 创建相应的线程(执行环境)；

- 提供相应的事件处理消息队列；
- 执行window的事件循环。

15.6 设备上下文

绘图是图形用户界面中最重要的一环。如前几节中所述，绘图是完全在客户端线程环境下完成，服务端线程中仅仅在必要的时候绘制窗口的标题和边框。

客户端绘图，需要在正确的地方绘制，例如当一个window在一个workbench之上时。当workbench需要更新，它不能够把window覆盖的区域给错误地绘成自己的图形。所以当一个客户端需要绘图时，它需要一个知道它当前所在区域的上下文环境，这个就是：(图形)设备上下文。



例如上图中的，workbench上的视图更新时，它不能在window覆盖的区域上进行更新，必须挖掉相应的区域，形成一个空洞。

注：视图去除掉window区域的操作一般图形用户界面中称之为剪切。

设备上下文(DC，Device Context)也可以认为是能够绘图的一个上下文环境，当前RT-Thread/GUI中包括两种DC：

- 硬件DC，即支持操作目标硬件的上下文环境
- 缓冲DC，即仅仅针对缓冲区进行绘图的上下文环境

其中，缓冲DC又能够通过blit（位元块传输）的方式刷新到目标硬件DC中。

创建两种DC的方式分别为：

```
struct rtgui_dc* rtgui_dc_buffer_create(int width, int height);
struct rtgui_dc* rtgui_dc_hw_create(rtgui_widget_t* owner);
```

在创建缓冲DC时，需要给定DC的宽度和高度，内部相应的分配和硬件DC相同色彩数的区域。例如，如果硬件DC是16色，即每一个像数点占用2字节以代表它的色彩，那么创建一个32x32的缓冲DC需要分配32x32x2=2048字节的像数缓冲区。

在创建硬件DC时，需要给出所属控件owner。同时为了方便控件绘图，额外提供了两个函数：

```
struct rtgui_dc* rtgui_dc_begin_drawing(rtgui_widget_t* owner);
void rtgui_dc_end_drawing(struct rtgui_dc* dc);
```

以表示控件上绘图的开始和结束(内部分别调用硬件DC创建和销毁函数)。

设备上下文的API

```
struct rtgui_dc* rtgui_dc_buffer_create(int width, int height);
rt_uint8_t* rtgui_dc_buffer_get_pixel(struct rtgui_dc* dc);
```

上面两个函数是与缓冲DC相关的，第一函数在上面已经有提及。rtgui_dc_buffer_get_pixel函数用于获得缓冲DC的像数缓冲区指针。

```
struct rtgui_dc* rtgui_dc_hw_create(rtgui_widget_t* owner);
struct rtgui_dc* rtgui_dc_begin_drawing(rtgui_widget_t* owner);
void rtgui_dc_end_drawing(struct rtgui_dc* dc);
```

上面几个函数是与硬件DC相关的，在上面已经有提及。

```
void rtgui_dc_destory(struct rtgui_dc* dc);
```

上面这个函数用于删除一个DC，并把释放相关的内存。这里的dc参数，可以是缓冲DC也可以是硬件DC。

```
void rtgui_dc_draw_point(struct rtgui_dc* dc, int x, int y);
void rtgui_dc_draw_vline(struct rtgui_dc* dc, int x, int y1, int y2);
void rtgui_dc_draw_hline(struct rtgui_dc* dc, int x1, int x2, int y);
void rtgui_dc_fill_rect (struct rtgui_dc* dc, struct rtgui_rect* rect);
void rtgui_dc_blit(struct rtgui_dc* dc, struct rtgui_point* dc_point, struct rtgui_dc* dest, rtgui_rect_t*
void rtgui_dc_draw_text (struct rtgui_dc* dc, const char* text, struct rtgui_rect* rect);
```

上面的API可以看成是DC绘图的核心API，其他API基本都是依赖于以上的API。

rtgui_dc_draw_point 用于在dc上的(x, y)坐标进行画点，而点的颜色即为dc当前的前景色。 rtgui_dc_draw_vline 用于在dc上画一条(x, y1) - (x, y2)的垂直线，线的颜色为dc当前的前景色。rtgui_dc_draw_hline 用于在dc上画一条(x1, y) - (x2, y)的水平线，线的颜色为dc当前的前景色。rtgui_dc_fill_rect 用于在dc上填充一个矩形框，矩形框的坐标信息由rect指定，颜色为dc当前的背景色。 rtgui_dc_draw_text 用于在dc上rect相应的矩形区域显示字符串。字符串颜色为dc当前的前景色，字符串对齐显示方式使用dc中文本对齐方式。

从上面的描述可以看出，默认参数中是不携带色彩参数的，而由dc当前的前景色和背景色决定。另外一个非常重要的，坐标总是相对于dc的坐标参考系，例如一个dc上绘制一个像素点：

```
1 rtgui_dc_t* dc;
2
3 /* 针对widget创建一个dc */
4 dc = rtgui_dc_begin_drawing(widget);
```

```
5
6 if (dc != RT_NULL)
7 {
8     rtgui_dc_draw_point(dc, 10, 10);
9 }
```

上面的例子代码中，会在(10, 10)坐标位置绘制一个当前前景色的像素点。其中的(10, 10)是相对于dc的坐标点，即相对于widget的坐标点。

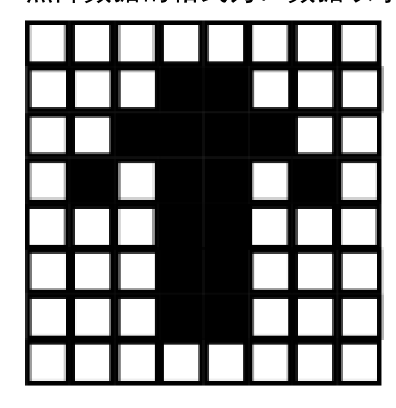
```
void rtgui_dc_draw_line (struct rtgui_dc* dc, int x1, int y1, int x2, int y2);
void rtgui_dc_draw_rect (struct rtgui_dc* dc, struct rtgui_rect* rect);
void rtgui_dc_draw_round_rect(struct rtgui_dc* dc, struct rtgui_rect* rect);
```

rtgui_dc_draw_line 用于在dc上(x1, y1) - (x2, y2)上绘一条线（斜线，水平线或垂直线），线的颜色是dc当前的前景色。 rtgui_dc_draw_rect 用于在dc上(x1, y1) - (x2, y2)区域上绘一个矩形框，线的颜色是dc当前的前景色。 rtgui_dc_draw_round_rect 用于在dc上(x1, y1) - (x2, y2)上绘一个圆角的矩形框，线的颜色是dc当前的前景色。

```
void rtgui_dc_draw_byte(struct rtgui_dc*dc, int x, int y, int h, const rt_uint8_t* data);
void rtgui_dc_draw_word(struct rtgui_dc*dc, int x, int y, int h, const rt_uint8_t* data);
```

rtgui_dc_draw_byte，rtgui_dc_draw_word是一类比较特殊的绘图操作，或者它们可以看成是显示点阵字体的最基本操作：绘制一个字符。这两个函数会在(x, y)坐标位置。h指示出点阵的高度，data指定存放点阵的数据。

点阵数据的格式为：数据以字节的格式存放，例如，下图的8x8的点阵，那么data数据应该是：



```
const rt_uint8_t data[] = {0x00, 0x18, 0x2c, 0x5a, 0x18, 0x18, 0x00};
```

rtgui_dc_draw_byte宽度是8，而rtgui_dc_draw_word宽度是16，即两个字节表示一行。

```
void rtgui_dc_draw_border(struct rtgui_dc* dc, rtgui_rect_t* rect, int flag);
void rtgui_dc_draw_horizontal_line(struct rtgui_dc* dc, int x1, int x2, int y);
void rtgui_dc_draw_vertical_line(struct rtgui_dc* dc, int x, int y1, int y2);
void rtgui_dc_draw_focus_rect(struct rtgui_dc* dc, rtgui_rect_t* rect);
```

rtgui_dc_draw_border 这个函数会根据flag的情况对矩形区域rect绘制边框，flag参数当前支持：

- RTGUI_BORDER_RAISE
- RTGUI_BORDER_SUNKEN

- RTGUI_BORDER_BOX
- RTGUI_BORDER_STATIC
- RTGUI_BORDER_EXTRA
- RTGUI_BORDER_SIMPLE

等几种边框。

rtgui_dc_draw_horizontal_line，rtgui_dc_draw_vertical_line函数会分别绘制一个立体的水平线和垂直线。 rtgui_dc_draw_focus_rect 函数会在指定的rect上绘制聚焦边框。

```
void rtgui_dc_draw_polygon(struct rtgui_dc* dc, const int *vx, const int *vy, int count);
void rtgui_dc_fill_polygon(struct rtgui_dc* dc, const int* vx, const int* vy, int count);

void rtgui_dc_draw_circle(struct rtgui_dc* dc, int x, int y, int r);
void rtgui_dc_fill_circle(struct rtgui_dc* dc, rt_int16_t x, rt_int16_t y, rt_int16_t r);
void rtgui_dc_draw_arc(struct rtgui_dc *dc, rt_int16_t x, rt_int16_t y, rt_int16_t r,
    rt_int16_t start, rt_int16_t end);

void rtgui_dc_draw_ellipse(struct rtgui_dc* dc, rt_int16_t x, rt_int16_t y, rt_int16_t rx, rt_int16_t ry);
void rtgui_dc_fill_ellipse(struct rtgui_dc *dc, rt_int16_t x, rt_int16_t y, rt_int16_t rx, rt_int16_t ry);
```

其中 rtgui_dc_draw_polygon，rtgui_dc_fill_polygon函数用于绘制多边形边框和填充多边形。 rtgui_dc_draw_circle，rtgui_dc_fill_circle函数用于绘制圆形和填充圆形。 rtgui_dc_draw_arc函数用于绘制圆弧。 rtgui_dc_draw_ellipse，rtgui_dc_fill_ellipse函数用于绘制椭圆和填充椭圆。

```
void rtgui_dc_set_color(struct rtgui_dc* dc, rtgui_color_t color);
rtgui_color_t rtgui_dc_get_color(struct rtgui_dc* dc);

void rtgui_dc_set_font(struct rtgui_dc* dc, rtgui_font_t* font);
rtgui_font_t* rtgui_dc_get_font(struct rtgui_dc* dc);
void rtgui_dc_set_textalign(struct rtgui_dc* dc, rt_int32_t align);
rt_int32_t rtgui_dc_get_textalign(struct rtgui_dc* dc);

rt_bool_t rtgui_dc_get_visible(struct rtgui_dc* dc);
void rtgui_dc_get_rect(struct rtgui_dc*dc, rtgui_rect_t* rect);
```

上面这些API主要用于设置/获取dc的一些当前属性，例如色彩，字体，文本对齐方式等等。

rtgui_dc_set/get_color用于设置/获取dc当前的前景色。 rtgui_dc_set/get_font用于设置/获取dc当前的字体。 rtgui_dc_set/get_textalign用于设置dc当前的文本对齐方式，当前版本支持如下对齐方式：

- RTGUI_ALIGN_LEFT
- RTGUI_ALIGN_RIGHT
- RTGUI_ALIGN_TOP
- RTGUI_ALIGN_BOTTOM
- RTGUI_ALIGN_CENTER_HORIZONTAL
- RTGUI_ALIGN_CENTER_VERTICAL

rtgui_dc_get_visible获得dc是否可见，当整个dc不可见(即隐藏状态)时，相应的绘图函数会自行跳过不进行绘图。 rtgui_dc_get_rect获得dc的矩形区域(相对于屏幕的绝对坐标)

简单的DC操作例程：

```
/*
 * 程序清单：DC操作演示
 *
 * 这个例子会在创建出的view上进行DC操作的演示
 */

#include "demo_view.h"
#include <rtgui/rtgui_system.h>
#include <rtgui/widgets/label.h>
#include <rtgui/widgets/slider.h>

/*
 * view的事件处理函数
 */
rt_bool_t dc_event_handler(rtgui_widget_t* widget, rtgui_event_t *event)
{
	/* 仅对PAINT事件进行处理 */
	if (event->type == RTGUI_EVENT_PAINT)
	{
		struct rtgui_dc* dc;
		rtgui_rect_t rect;
		rt_uint32_t vx[] = {20, 50, 60, 45, 60, 20};
		rt_uint32_t vy[] = {150, 50, 90, 60, 45, 50};

		/*
		 * 因为用的是demo view，上面本身有一部分控件，所以在绘图时先要让demo view
		 * 先绘图
		 */
		rtgui_view_event_handler(widget, event);

		/************************************************************************/
		/* 下面的是DC的操作                                                     */
		/************************************************************************/

		/* 获得控件所属的DC */
		dc = rtgui_dc_begin_drawing(widget);
		/* 如果不能正常获得DC，返回（如果控件或父控件是隐藏状态，DC是获取不成功的） */
		if (dc == RT_NULL)
			return RT_FALSE;

		/* 获得demo view允许绘图的区域 */
		demo_view_get_rect(RTGUI_VIEW(widget), &rect);

		rtgui_dc_set_textalign(dc, RTGUI_ALIGN_BOTTOM | RTGUI_ALIGN_CENTER_HORIZONTAL);
		/* 显示GUI的版本信息 */
#ifdef RTGUI_USING_SMALL_SIZE
		rtgui_dc_draw_text(dc, "RT-Thread/GUI小型版本", &rect);
#else
		rtgui_dc_draw_text(dc, "RT-Thread/GUI标准版本", &rect);
```

```
50 #endif
51
52             /* 绘制一个圆形 */
53             rtgui_dc_set_color(dc, red);
54             rtgui_dc_draw_circle(dc, rect.x1 + 10, rect.y1 + 10, 10);
55
56             /* 填充一个圆形 */
57             rtgui_dc_set_color(dc, green);
58             rtgui_dc_fill_circle(dc, rect.x1 + 30, rect.y1 + 10, 10);
59
60             /* 画一个圆弧 */
61             rtgui_dc_set_color(dc, RTGUI_RGB(250, 120, 120));
62             rtgui_dc_draw_arc(dc, rect.x1 + 120, rect.y1 + 60, 30, 0, 120);
63
64             /* 多边形 */
65             rtgui_dc_set_color(dc, blue);
66             rtgui_dc_draw_polygon(dc, vx, vy, 6);
67
68             /* 绘制不同的边框 */
69             {
70                     rtgui_rect_t rect = {0, 0, 16, 16};
71                     rtgui_rect_moveto(&rect, 30, 120);
72
73                     rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_RAISE);
74                     rect.x1 += 20;
75                     rect.x2 += 20 + 50;
76                     rtgui_dc_draw_text(dc, "raise", &rect);
77                     rect.x1 -= 20;
78                     rect.x2 -= 20 + 50;
79                     rect.y1 += 20;
80                     rect.y2 += 20;
81
82                     rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_SIMPLE);
83                     rect.x1 += 20;
84                     rect.x2 += 20 + 50;
85                     rtgui_dc_draw_text(dc, "simple", &rect);
86                     rect.x1 -= 20;
87                     rect.x2 -= 20 + 50;
88                     rect.y1 += 20;
89                     rect.y2 += 20;
90
91                     rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_SUNKEN);
92                     rect.x1 += 20;
93                     rect.x2 += 20 + 50;
94                     rtgui_dc_draw_text(dc, "sunken", &rect);
95                     rect.x1 -= 20;
96                     rect.x2 -= 20 + 50;
97                     rect.y1 += 20;
98                     rect.y2 += 20;
99
100                    rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_BOX);
101                    rect.x1 += 20;
102                    rect.x2 += 20 + 50;
```

```
103                         rtgui_dc_draw_text(dc, "box", &rect);
104                         rect.x1 -= 20;
105                         rect.x2 -= 20 + 50;
106                         rect.y1 += 20;
107                         rect.y2 += 20;
108 
109                         rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_STATIC);
110                         rect.x1 += 20;
111                         rect.x2 += 20 + 50;
112                         rtgui_dc_draw_text(dc, "static", &rect);
113                         rect.x1 -= 20;
114                         rect.x2 -= 20 + 50;
115                         rect.y1 += 20;
116                         rect.y2 += 20;
117 
118                         rtgui_dc_draw_border(dc, &rect, RTGUI_BORDER_EXTRA);
119                         rect.x1 += 20;
120                         rect.x2 += 20 + 50;
121                         rtgui_dc_draw_text(dc, "extra", &rect);
122                         rect.x1 -= 20;
123                         rect.x2 -= 20 + 50;
124                         rect.y1 += 20;
125                         rect.y2 += 20;
126                 }
127 
128                 /* 绘图完成 */
129                 rtgui_dc_end_drawing(dc);
130         }
131         else
132         {
133                 /* 其他事件，调用默认的事件处理函数 */
134                 return rtgui_view_event_handler(widget, event);
135         }
136 
137         return RT_FALSE;
138 }
139 
140 /* 创建用于DC操作演示用的视图 */
141 rtgui_view_t *demo_view_dc(rtgui_workbench_t* workbench)
142 {
143         rtgui_view_t *view;
144 
145         view = demo_view(workbench, "DC Demo");
146         if (view != RT_NULL)
147                 /* 设置成自己的事件处理函数 */
148                 rtgui_widget_set_event_handler(RTGUI_WIDGET(view), dc_event_handler);
149 
150         return view;
151 }
```

在DC上显示一副图像的例程：

```c
/*
 * 程序清单：DC上显示图像演示
 *
 * 这个例子会在创建出的view上显示图像
 */

#include "demo_view.h"
#include <rtgui/widgets/button.h>
#include <rtgui/widgets/filelist_view.h>

static rtgui_image_t* image = RT_NULL;
static rtgui_view_t* _view = RT_NULL;

/* 打开按钮的回调函数 */
static void open_btn_onbutton(rtgui_widget_t* widget, struct rtgui_event* event)
{
        rtgui_filelist_view_t *view;
        rtgui_workbench_t *workbench;
        rtgui_rect_t rect;

        /* 获得顶层的workbench */
        workbench = RTGUI_WORKBENCH(rtgui_widget_get_toplevel(widget));
        rtgui_widget_get_rect(RTGUI_WIDGET(workbench), &rect);

        /* WIN32平台上和真实设备上的初始路径处理 */
#ifdef _WIN32
        view = rtgui_filelist_view_create(workbench, "d:\\", "*.*", &rect);
#else
        view = rtgui_filelist_view_create(workbench, "/", "*.*", &rect);
#endif
        /* 模态显示一个文件列表视图，以提供给用户选择图像文件 */
        if (rtgui_view_show(RTGUI_VIEW(view), RT_TRUE) == RTGUI_MODAL_OK)
        {
                char path[32], image_type[8];

                /* 设置文件路径的标签 */
                rtgui_filelist_get_fullpath(view, path, sizeof(path));
                if (image != RT_NULL) rtgui_image_destroy(image);

                rt_memset(image_type, 0, sizeof(image_type));

                /* 获得图像的类型 */
                if (rt_strstr(path, ".png") != RT_NULL ||
                        rt_strstr(path, ".PNG") != RT_NULL)
                        strcat(image_type, "png");
                if (rt_strstr(path, ".jpg") != RT_NULL ||
                        rt_strstr(path, ".JPG") != RT_NULL)
                        strcat(image_type, "jpeg");
                if (rt_strstr(path, ".hdc") != RT_NULL ||
                        rt_strstr(path, ".HDC") != RT_NULL)
                        strcat(image_type, "hdc");

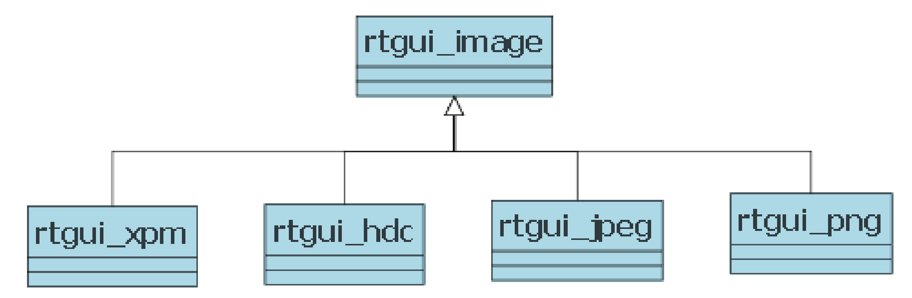
```

```
53                      /* 如果图像文件有效，创建相应的rtgui_image对象 */
54                      if (image_type[0] != '\0')
55                              image = rtgui_image_create_from_file(image_type, path, RT_TRUE);
56              }
57
58              /* 删除 文件列表 视图 */
59              rtgui_view_destroy(RTGUI_VIEW(view));
60              rtgui_view_show(_view, RT_FALSE);
61      }
62
63      /* 演示视图的事件处理函数 */
64      static rt_bool_t demo_view_event_handler(rtgui_widget_t* widget, rtgui_event_t *event)
65      {
66              rt_bool_t result;
67
68              /* 先调用默认的事件处理函数(这里只关心PAINT事件，但演示视图还有本身的一些控件) */
69              result = rtgui_view_event_handler(widget, event);
70
71              if (event->type == RTGUI_EVENT_PAINT)
72              {
73                      struct rtgui_dc* dc;
74                      rtgui_rect_t rect;
75
76                      /* 获得控件所属的DC */
77                      dc = rtgui_dc_begin_drawing(widget);
78                      if (dc == RT_NULL)
79                              /* 如果不能正常获得DC，返回(如果控件或父控件是隐藏状态，DC是获取不成功的) */
80                              return RT_FALSE;
81
82                      /* 获得demo view允许绘图的区域 */
83                      demo_view_get_rect(RTGUI_VIEW(widget), &rect);
84
85                      /* 获得图像显示区域 */
86                      rect.x1 += 5; rect.x2 -= 5;
87                      rect.y1 += 30;
88
89                      if (image != RT_NULL)
90                              rtgui_image_blit(image, dc, &rect);
91
92                      /* 绘图完成 */
93                      rtgui_dc_end_drawing(dc);
94              }
95
96              return result;
97      }
98
99      /* 创建用于显示图像的演示视图 */
100     rtgui_view_t* demo_view_image(rtgui_workbench_t* workbench)
101     {
102             rtgui_rect_t rect;
103             rtgui_button_t* open_btn;
104
105             /* 先创建一个演示视图 */
```

```
106         _view = demo_view(workbench, "图像演示");
107         if (_view != RT_NULL)
108                 /* 设置默认的事件处理函数到demo_view_event_handler函数 */
109                 rtgui_widget_set_event_handler(RTGUI_WIDGET(_view), demo_view_event_handler);
110 
111         /* 添加一个按钮 */
112         demo_view_get_rect(_view, &rect);
113         rect.x1 += 5; rect.x2 = rect.x1 + 120;
114         rect.y2 = rect.y1 + 20;
115         open_btn = rtgui_button_create("打开图像文件");
116         rtgui_container_add_child(RTGUI_CONTAINER(_view), RTGUI_WIDGET(open_btn));
117         rtgui_widget_set_rect(RTGUI_WIDGET(open_btn), &rect);
118         rtgui_button_set_onbutton(open_btn, open_btn_onbutton);
119 
120         return _view;
121 }
```

15.7 图像引擎

RT-Thread/GUI的各种图像格式支持是采用C语言的面向对象方式实现的，如下图所示：



目前RTGUI支持四种格式：XPM，HDC，JPEG和PNG格式。rtgui_image是他们的基类，对于上层提供对各种图像格式都适用的操作接口。

15.7.1 rtgui_image基类

```
struct rtgui_image
{
        /* 图像大小信息 */
        rt_uint16_t w, h;
        /* 图形引擎 */
        struct rtgui_image_engine* engine;
        /* 图像私有数据 */
        void* data;
};
```

rtgui_image成员中包括了

名称	描述
w	图像宽度，单位为像素
h	图像高度，单位为像素
engine	图像实际使用的引擎，不同的图像格式包括不同的图像引擎
data	私有数据，此数据的具体格式由不同的图像引擎自行解析。

由上面的成员结构可以看出，其中的engine和data是由不同的图像格式所拥有的，而要实现不同的图像格式支持就需要用好这两个成员变量。

15.7.2 rtgui_image 引擎接口

图像引擎接口即rtgui_image_engine结构体定义的，它们是一些公共的数据访问函数（接口），具体定义如下：

```
struct rtgui_image_engine
{
        /* 引擎名称 */
        const char* name;
        struct rtgui_list_node list;

        /* 图像引擎接口函数 */

        /* 对于一指定文件句柄，判断是否是相应的图像格式 */
        rt_bool_t (*image_check)(struct rtgui_filerw* file);
        /* 根据文件句柄载入图像数据 */
        rt_bool_t (*image_load)(struct rtgui_image* image, struct rtgui_filerw* file, rt_bool_t load);
        /* 卸载图像数据 */
        void (*image_unload)(struct rtgui_image* image);
        /* 对目标dc做绘图 */
        void (*image_blit)(struct rtgui_image* image, struct rtgui_dc* dc, struct rtgui_rect* rect);
};
```

每个接口的意义如下：

名称	描述
image_check	根据指定的文件句柄，判断是否是相应的图像格式，是则返回RT_TRUE;
image_load	根据文件句柄载入图像数据，load参数指示出是否全部载入图像数据。
image_unload	卸载图像数据
image_blit	对目标dc做绘图，参数rect指示出目标DC中图像部分的矩形区域。

创建一个rtgui_image对象的接口分为两类：

`rtgui_image*` **`rtgui_image_create_from_file`**(*const char* type, const char* filename, rt_bool_t load*)

这个函数用于从文件中创建一个图像类型为 type 的rtgui_image对象。

`rtgui_image*` **`rtgui_image_create_from_mem`**(*const char* type, const rt_uint8_t* data, rt_size_t length, rt_bool_t load*)

这个函数用于从内存数据块中创建一个图像类型为 type 的rtgui_image对象。

函数接口：

`void` **`rtgui_image_destroy`**(*struct rtgui_image* image*)

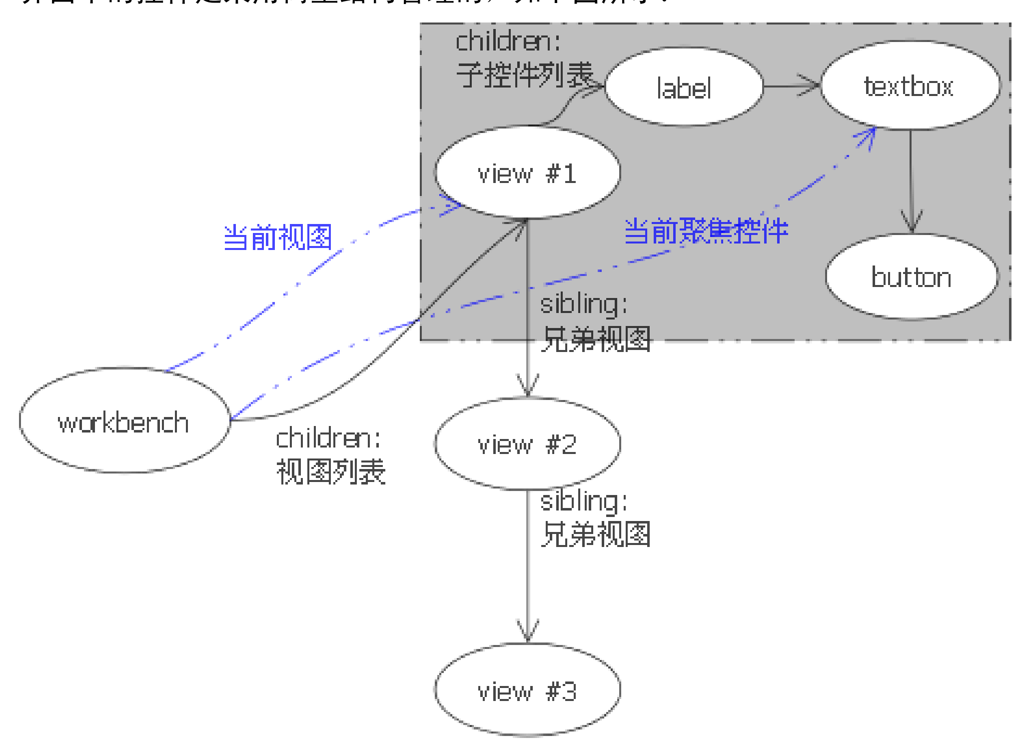
用于释放一个rtgui_image对象。在这个函数中，它将调用rt_bool_t (*image_unload)(struct rtgui_image* image)函数进行图像格式私有数据的卸载。

15.7.3 添加图像格式

添加新图像格式支持需要实现上节中指出的图像引擎接口，同时下面几点应该是实现时需要注意的地方：
载入图像时的参数load，如果为RT_TRUE，那么表示全部载入到内存中进行图像解码，进行图像绘制的时候就不需要再做图像解码的工作。如果是 RT_FALSE，那么表示开始时并不载入到内存中，只有当要进行图像绘制时才进行图像解码。前者相对来说绘图的速度比较快，但比较占用内存。 # 卸载图像时，需要释放图像格式的私有数据。 # 图像格式的私有数据可以保存在image->data成员上，需要自行转换成自己的数据结构体进行解释。

15.8 控件树结构及事件派发

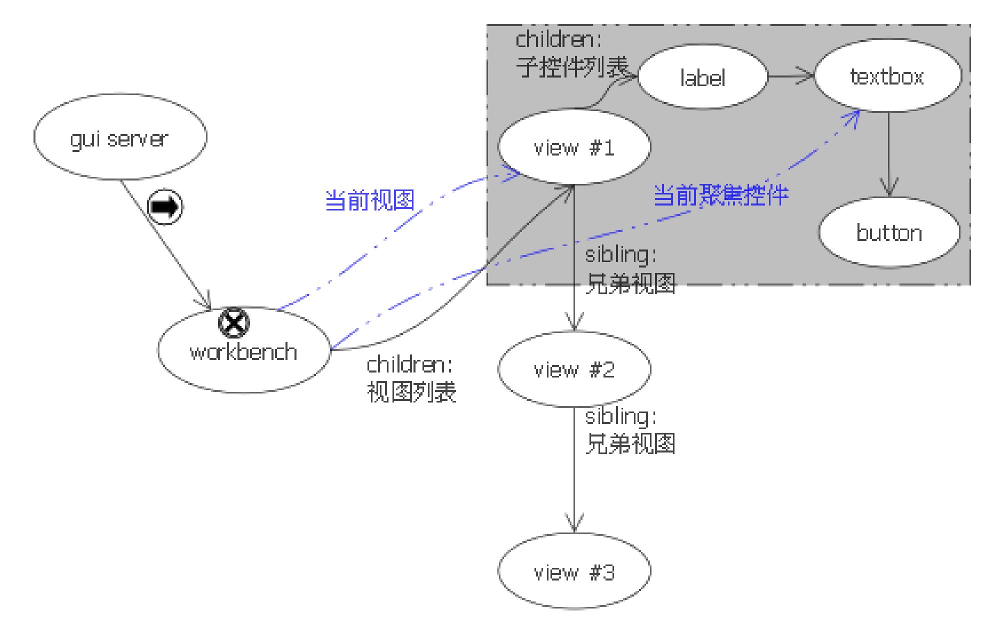
界面中的控件是采用树型结构管理的，如下图所示：



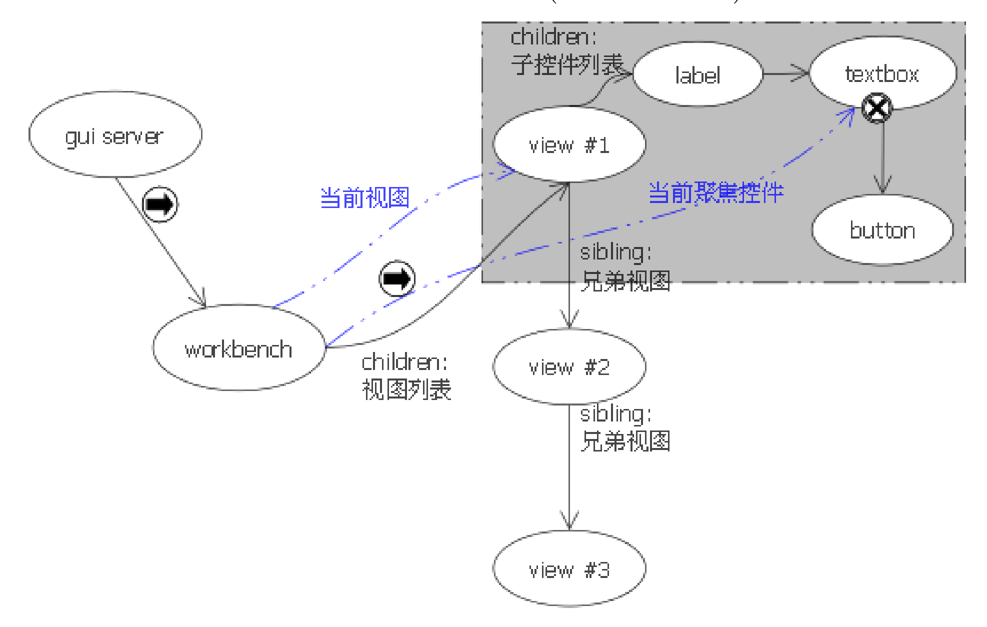
上图所示的界面中，其组织结构如下：运行时处于顶层的是workbench控件对象或window控件对象，这两个类都是从rtgui_toplevel_t类派生的，并且都是容器类。对于一个容器类，其中包含了一个children列表用于链接各个子控件，在上面的图中包括了view #1、view #2、view #3的列表(在workbench中，其直接底层子控件只能包括视图控件，而window中则不受这个限制)。view #1、view #2、view #3则依赖于sibling链接形成兄弟节点(view类派生自rtgui_widget_t类，每个rtgui_widget_t类中都包含一个sibling节点)。同理，上图中的label对象，textbox对象和button依赖于sibling节点形成一个链表并挂接在view #1对象的children链表头上。通过children节点和sibling节点，每个toplevel对象形成了一个以toplevel节点为根节点的树型结构。

RT-Thread/GUI根据其事件消息的不同分成了几类：

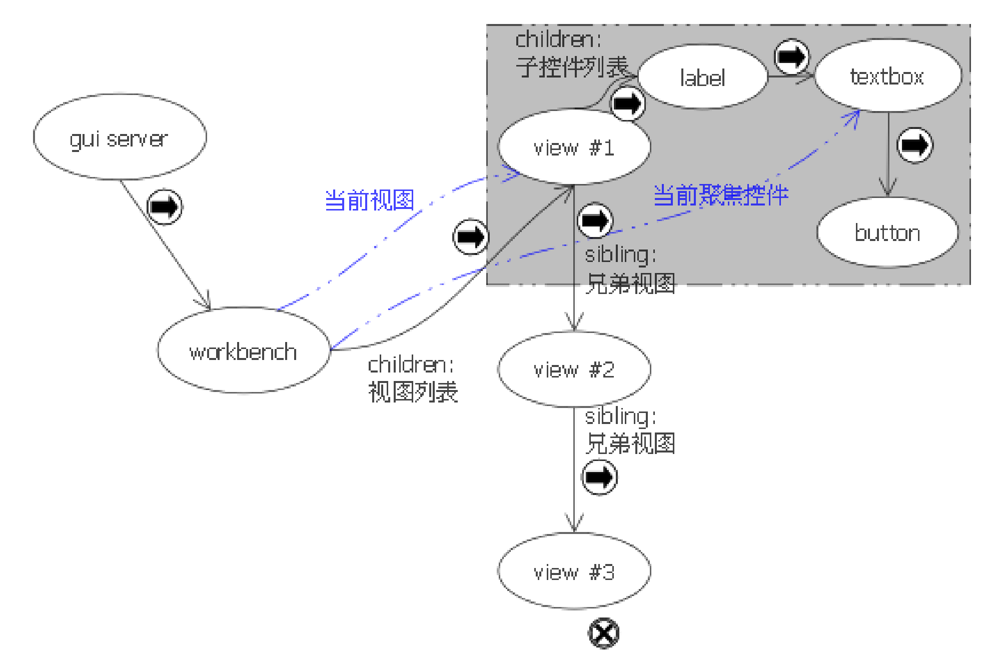
- 服务端事件终结于workbench、window对象上。如下图所示：



- 服务端事件终结于当前获得焦点的控件对象上(例如键盘事件)。如下图所示：



- 服务端事件广播到控件树所有节点控件对象上(例如服务端发送过来的PAINT事件，用于重绘一颗完整控件树)。如下图所示：



当服务端事件发送到workbench时，workbench会遵循如下的原则进行事件派发(事件广播的情况)：调用event_handler虚拟函数处理事件，如果这个虚拟函数返回RT_TRUE，那么事件将不进行下一步控件传递。如果事件处理器返回值是RT_FALSE，事件将沿着sibling节点的方向继续进行处理。

15.9 系统配置与图形驱动

15.9.1 系统配置

RT-Thread/GUI的配置通常依赖于rtgui_config.h头文件，但为了与RT-Thread实时操作系统的配置文件整合起来，可以在它的配置头文件rtconfig.h中定义相应的选项。

```
/* SECTION: RT-Thread/GUI, 定义RT_USING_RTGUI以在RT-Thread中使用GUI */
#define RT_USING_RTGUI

/* 一个RT-Thread/GUI对象的名字长度，如果长度过长，每个客户端线程的单条消息长度也相应的加大 */
#define RTGUI_NAME_MAX       12
/* 支持16点阵字体 */
#define RTGUI_USING_FONT16
/* 支持中文字体 */
#define RTGUI_USING_FONTHZ
/* 使用DFS文件系统作为文件操作接口 */
#define RTGUI_USING_DFS_FILERW
/* 中文字体使用字体文件，需放在/resource目录下 */
#define RTGUI_USING_HZ_FILE
/* 使用RT-Thread/GUI小型版本 */
#define RTGUI_USING_SMALL_SIZE
/* 不使用鼠标 */
/* #define RTGUI_USING_MOUSE_CURSOR */
/* 默认的字体大小 */
#define RTGUI_DEFAULT_FONT_SIZE 16
/* 不使用内嵌中文字体 */
```

```
/* #define RTGUI_USING_HZ_BMP */
/* 不支持PNG图像 */
/* #define RTGUI_IMAGE_PNG */
/* 不支持JPEG图像 */
/* #define RTGUI_IMAGE_JPEG */
```

其他rtgui_config.h中的选项包括：

```
/* GUI服务端线程优先级，时间片大小和栈大小 */
#define RTGUI_SVR_THREAD_PRIORITY        15
#define RTGUI_SVR_THREAD_TIMESLICE       5
#define RTGUI_SVR_THREAD_STACK_SIZE      1024

/* GUI客户端线程优先级，时间片大小和栈大小 */
#define RTGUI_APP_THREAD_PRIORITY        25
#define RTGUI_APP_THREAD_TIMESLICE       5
#define RTGUI_APP_THREAD_STACK_SIZE      1024
```

15.9.2 图形驱动

图形驱动是RT-Thread/GUI最底层的接口，是上层应用绘图操作的最终反映。一个图形驱动需要遵循如下接口：

```
struct rtgui_graphic_driver
{
    /* 驱动名称 */
    char* name;

    /* 每个像素占用的字节数 */
    rt_uint16_t byte_per_pixel;

    /* 屏幕的宽和高(以像素为单位) */
    rt_uint16_t width;
    rt_uint16_t height;

    /* 屏幕更新操作 */
    void (*screen_update)(rtgui_rect_t* rect);

    /* 获得硬件视频帧缓冲指针 */
    rt_uint8_t* (*get_framebuffer)(void);

    /* 在屏幕的(x, y)坐标位置绘制一个像素点 */
    void (*set_pixel) (rtgui_color_t *c, rt_base_t x, rt_base_t y);
    /* 在屏幕的(x, y)坐标位置读取一个像素点 */
    void (*get_pixel) (rtgui_color_t *c, rt_base_t x, rt_base_t y);

    /* 在屏幕的(x1,y) - (x2, y)位置绘制一条水平线 */
    void (*draw_hline)(rtgui_color_t *c, rt_base_t x1, rt_base_t x2, rt_base_t y);
    /* 在屏幕的(x,y1) - (x, y2)位置绘制一条垂直线 */
    void (*draw_vline)(rtgui_color_t *c, rt_base_t x , rt_base_t y1, rt_base_t y2);
```

```
    /* 在屏幕的(x1,y) - (x2, y)位置绘制一条原始水平线 */
    void (*draw_raw_hline)(rt_uint8_t *pixels, rt_base_t x1, rt_base_t x2, rt_base_t y);

    /* 指向下一个驱动的列表 */
    rtgui_list_t list;
};
```

其中这个结构体中的函数指针:

```
/* 屏幕更新操作 */
void (*screen_update)(rtgui_rect_t* rect);

/* 获得硬件视频帧缓冲指针 */
rt_uint8_t* (*get_framebuffer)(void);
```

并不是必须的。函数 screen_update 用于在绘图完成后，需要统一刷新到硬件中的情况。get_framebuffer 用于具备frame buffer的硬件中。没有get_framebuffer 的实现不会对系统照成大的影响，但如果有这个实现将能够加快绘图速度。

函数指针:

```
/* 在屏幕的(x1,y) - (x2, y)位置绘制一条原始水平线 */
void (*draw_raw_hline)(rt_uint8_t *pixels, rt_base_t x1, rt_base_t x2, rt_base_t y);
```

用于快速的绘制一条水平线，pixels中保存着像素点数据。

一个完整的图形驱动（用于STM32上）:

```
#include <rtthread.h>
#include "stm3210c_eval_lcd.h"
#include "stm32f10x.h"
#include "stm32f10x_spi.h"

#include <rtgui/rtgui.h>
#include <rtgui/driver.h>
#include <rtgui/rtgui_system.h>
#include <rtgui/rtgui_server.h>

#define START_BYTE      0x70
#define SET_INDEX       0x00
#define READ_STATUS     0x01
#define LCD_WRITE_REG   0x02
#define LCD_READ_REG    0x03

/* RT-Thread/GUI接口函数的声明 */
void rt_hw_lcd_update(rtgui_rect_t *rect);
rt_uint8_t * rt_hw_lcd_get_framebuffer(void);
void rt_hw_lcd_set_pixel(rtgui_color_t *c, rt_base_t x, rt_base_t y);
void rt_hw_lcd_get_pixel(rtgui_color_t *c, rt_base_t x, rt_base_t y);
void rt_hw_lcd_draw_hline(rtgui_color_t *c, rt_base_t x1, rt_base_t x2, rt_base_t y);
void rt_hw_lcd_draw_vline(rtgui_color_t *c, rt_base_t x, rt_base_t y1, rt_base_t y2);
void rt_hw_lcd_draw_raw_hline(rt_uint8_t *pixels, rt_base_t x1, rt_base_t x2, rt_base_t y);
```

```
/* 驱动结构体的初始化 */
struct rtgui_graphic_driver _rtgui_lcd_driver =
{
    "lcd",
    2,
    320,
    240,
    rt_hw_lcd_update,
    rt_hw_lcd_get_framebuffer,
    rt_hw_lcd_set_pixel,
    rt_hw_lcd_get_pixel,
    rt_hw_lcd_draw_hline,
    rt_hw_lcd_draw_vline,
    rt_hw_lcd_draw_raw_hline
};

static void _delay_(__IO uint32_t nCount)
{
    __IO uint32_t index = 0;
    for(index = (100000 * nCount); index != 0; index--)
    {}
}

/**
  * @brief  Sets or reset LCD control lines.
  * @param  GPIOx: where x can be B or D to select the GPIO peripheral.
  * @param  CtrlPins: the Control line. This parameter can be:
  *     @arg LCD_NCS_PIN: Chip Select pin
  * @param  BitVal: specifies the value to be written to the selected bit.
  *   This parameter can be:
  *     @arg Bit_RESET: to clear the port pin
  *     @arg Bit_SET: to set the port pin
  * @retval None
  */
void LCD_CtrlLinesWrite(GPIO_TypeDef* GPIOx, uint16_t CtrlPins, BitAction BitVal)
{
    /* Set or Reset the control line */
    GPIO_WriteBit(GPIOx, CtrlPins, BitVal);
}

/**
  * @brief  Reset LCD control line(/CS) and Send Start-Byte
  * @param  Start_Byte: the Start-Byte to be sent
  * @retval None
  */
void LCD_nCS_StartByte(uint8_t Start_Byte)
{
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_RESET);
    SPI_I2S_SendData(LCD_SPI, Start_Byte);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
}
```

```
/**
  * @brief  Configures LCD control lines in Output Push-Pull mode.
  * @param  None
  * @retval None
  */
void LCD_CtrlLinesConfig(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable GPIO clock */
    RCC_APB2PeriphClockCmd(LCD_NCS_GPIO_CLK, ENABLE);

    /* Configure NCS in Output Push-Pull mode */
    GPIO_InitStructure.GPIO_Pin = LCD_NCS_PIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(LCD_NCS_GPIO_PORT, &GPIO_InitStructure);
}

/**
  * @brief  Writes index to select the LCD register.
  * @param  LCD_Reg: address of the selected register.
  * @retval None
  */
void LCD_WriteRegIndex(uint8_t LCD_Reg)
{
    /* Reset LCD control line(/CS) and Send Start-Byte */
    LCD_nCS_StartByte(START_BYTE | SET_INDEX);
    /* Write 16-bit Reg Index (High Byte is 0) */
    SPI_I2S_SendData(LCD_SPI, 0x00);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    SPI_I2S_SendData(LCD_SPI, LCD_Reg);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
}

/**
  * @brief  Reads the selected LCD Register.
  * @param  None
  * @retval LCD Register Value.
  */
uint16_t LCD_ReadReg(uint8_t LCD_Reg)
{
    uint16_t tmp = 0;
    uint8_t i = 0;

    /* LCD_SPI prescaler: 4 */
    LCD_SPI->CR1 &= 0xFFC7;
    LCD_SPI->CR1 |= 0x0008;
    /* Write 16-bit Index (then Read Reg) */
    LCD_WriteRegIndex(LCD_Reg);
```

```
    /* Read 16-bit Reg */
    /* Reset LCD control line(/CS) and Send Start-Byte */
    LCD_nCS_StartByte(START_BYTE | LCD_READ_REG);

    for(i = 0; i < 5; i++)
    {
        SPI_I2S_SendData(LCD_SPI, 0xFF);
        while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
        {}
        /* One byte of invalid dummy data read after the start byte */
        while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_RXNE) == RESET)
        {}
        SPI_I2S_ReceiveData(LCD_SPI);
    }
    SPI_I2S_SendData(LCD_SPI, 0xFF);
    /* Read upper byte */
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    /* Read lower byte */
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_RXNE) == RESET)
    {}
    tmp = SPI_I2S_ReceiveData(LCD_SPI);


    SPI_I2S_SendData(LCD_SPI, 0xFF);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    /* Read lower byte */
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_RXNE) == RESET)
    {}
    tmp = ((tmp & 0xFF) << 8) | SPI_I2S_ReceiveData(LCD_SPI);
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
    /* LCD_SPI prescaler: 2 */
    LCD_SPI->CR1 &= 0xFFC7;
    return tmp;
}

/**
  * @brief  Writes to the selected LCD register.
  * @param  LCD_Reg: address of the selected register.
  * @param  LCD_RegValue: value to write to the selected register.
  * @retval None
  */
void LCD_WriteReg(uint8_t LCD_Reg, uint16_t LCD_RegValue)
{
    /* Write 16-bit Index (then Write Reg) */
    LCD_WriteRegIndex(LCD_Reg);
    /* Write 16-bit Reg */
    /* Reset LCD control line(/CS) and Send Start-Byte */
    LCD_nCS_StartByte(START_BYTE | LCD_WRITE_REG);
    SPI_I2S_SendData(LCD_SPI, LCD_RegValue>>8);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
```

```
    SPI_I2S_SendData(LCD_SPI, (LCD_RegValue & 0xFF));
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
}

/**
  * @brief  Writes to the LCD RAM.
  * @param  RGB_Code: the pixel color in RGB mode (5-6-5).
  * @retval None
  */
void LCD_WriteRAM(uint16_t RGB_Code)
{
    SPI_I2S_SendData(LCD_SPI, RGB_Code >> 8);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
    SPI_I2S_SendData(LCD_SPI, RGB_Code & 0xFF);
    while(SPI_I2S_GetFlagStatus(LCD_SPI, SPI_I2S_FLAG_BSY) != RESET)
    {}
}

/**
  * @brief  Prepare to write to the LCD RAM.
  * @param  None
  * @retval None
  */
void LCD_WriteRAM_Prepare(void)
{
    LCD_WriteRegIndex(R34); /* Select GRAM Reg */
    /* Reset LCD control line(/CS) and Send Start-Byte */
    LCD_nCS_StartByte(START_BYTE | LCD_WRITE_REG);
}

/**
  * @brief  Writes 1 word to the LCD RAM.
  * @param  RGB_Code: the pixel color in RGB mode (5-6-5).
  * @retval None
  */
void LCD_WriteRAMWord(uint16_t RGB_Code)
{
    LCD_WriteRAM_Prepare();
    LCD_WriteRAM(RGB_Code);
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
}

/**
  * @brief  Power on the LCD.
  * @param  None
  * @retval None
  */
void LCD_PowerOn(void)
{
    /* Power On sequence ---------------------------------------------------------*/
```

```
    LCD_WriteReg(R16, 0x0000); /* SAP, BT[3:0], AP, DSTB, SLP, STB */
    LCD_WriteReg(R17, 0x0000); /* DC1[2:0], DC0[2:0], VC[2:0] */
    LCD_WriteReg(R18, 0x0000); /* VREG1OUT voltage */
    LCD_WriteReg(R19, 0x0000); /* VDV[4:0] for VCOM amplitude */
    _delay_(20);                /* Dis-charge capacitor power voltage (200ms) */
    LCD_WriteReg(R16, 0x17B0); /* SAP, BT[3:0], AP, DSTB, SLP, STB */
    LCD_WriteReg(R17, 0x0137); /* DC1[2:0], DC0[2:0], VC[2:0] */
    _delay_(5);                 /* Delay 50 ms */
    LCD_WriteReg(R18, 0x0139); /* VREG1OUT voltage */
    _delay_(5);                 /* delay 50 ms */
    LCD_WriteReg(R19, 0x1d00); /* VDV[4:0] for VCOM amplitude */
    LCD_WriteReg(R41, 0x0013); /* VCM[4:0] for VCOMH */
    _delay_(5);                 /* delay 50 ms */
    LCD_WriteReg(R7, 0x0173);  /* 262K color and display ON */
}

/**
  * @brief  Enables the Display.
  * @param  None
  * @retval None
  */
void LCD_DisplayOn(void)
{
    /* Display On */
    LCD_WriteReg(R7, 0x0173); /* 262K color and display ON */
}

/**
  * @brief  Disables the Display.
  * @param  None
  * @retval None
  */
void LCD_DisplayOff(void)
{
    /* Display Off */
    LCD_WriteReg(R7, 0x0);
}

/**
  * @brief  Configures the LCD_SPI interface.
  * @param  None
  * @retval None
  */
void LCD_SPIConfig(void)
{
    SPI_InitTypeDef    SPI_InitStructure;
    GPIO_InitTypeDef   GPIO_InitStructure;

    /* Enable GPIO clock */
    RCC_APB2PeriphClockCmd(LCD_SPI_GPIO_CLK | RCC_APB2Periph_AFIO, ENABLE);
    GPIO_PinRemapConfig(GPIO_Remap_SPI3, ENABLE);

    /* Enable SPI clock  */
```

```
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI3, ENABLE);

    /* Configure SPI pins: SCK, MISO and MOSI */
    GPIO_InitStructure.GPIO_Pin = LCD_SPI_SCK_PIN | LCD_SPI_MISO_PIN | LCD_SPI_MOSI_PIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(LCD_SPI_GPIO_PORT, &GPIO_InitStructure);

    SPI_I2S_DeInit(LCD_SPI);

    /* SPI Config */
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_Init(LCD_SPI, &SPI_InitStructure);

    /* SPI enable */
    SPI_Cmd(LCD_SPI, ENABLE);
}

/**
  * @brief  Setups the LCD.
  * @param  None
  * @retval None
  */
void LCD_Setup(void)
{
    /* Configure the LCD Control pins ----------------------------------------*/
    LCD_CtrlLinesConfig();

    /* Configure the LCD_SPI interface ---------------------------------------*/
    LCD_SPIConfig();
    _delay_(5); /* Delay 50 ms */
    /* Start Initial Sequence ------------------------------------------------*/
    LCD_WriteReg(R229, 0x8000); /* Set the internal vcore voltage */
    LCD_WriteReg(R0,  0x0001); /* Start internal OSC. */
    LCD_WriteReg(R1,  0x0100); /* set SS and SM bit */
    LCD_WriteReg(R2,  0x0700); /* set 1 line inversion */
    LCD_WriteReg(R3,  0x1030); /* set GRAM write direction and BGR=1. */
    LCD_WriteReg(R4,  0x0000); /* Resize register */
    LCD_WriteReg(R8,  0x0202); /* set the back porch and front porch */
    LCD_WriteReg(R9,  0x0000); /* set non-display area refresh cycle ISC[3:0] */
    LCD_WriteReg(R10, 0x0000); /* FMARK function */
    LCD_WriteReg(R12, 0x0000); /* RGB interface setting */
    LCD_WriteReg(R13, 0x0000); /* Frame marker Position */
    LCD_WriteReg(R15, 0x0000); /* RGB interface polarity */
    /* Power On sequence -----------------------------------------------------*/
    LCD_WriteReg(R16, 0x0000); /* SAP, BT[3:0], AP, DSTB, SLP, STB */
```

```
LCD_WriteReg(R17, 0x0000); /* DC1[2:0], DC0[2:0], VC[2:0] */
LCD_WriteReg(R18, 0x0000); /* VREG1OUT voltage */
LCD_WriteReg(R19, 0x0000); /* VDV[4:0] for VCOM amplitude */
_delay_(20);                /* Dis-charge capacitor power voltage (200ms) */
LCD_WriteReg(R16, 0x17B0); /* SAP, BT[3:0], AP, DSTB, SLP, STB */
LCD_WriteReg(R17, 0x0137); /* DC1[2:0], DC0[2:0], VC[2:0] */
_delay_(5);                 /* Delay 50 ms */
LCD_WriteReg(R18, 0x0139); /* VREG1OUT voltage */
_delay_(5);                 /* Delay 50 ms */
LCD_WriteReg(R19, 0x1d00); /* VDV[4:0] for VCOM amplitude */
LCD_WriteReg(R41, 0x0013); /* VCM[4:0] for VCOMH */
_delay_(5);                 /* Delay 50 ms */
LCD_WriteReg(R32, 0x0000); /* GRAM horizontal Address */
LCD_WriteReg(R33, 0x0000); /* GRAM Vertical Address */
/* Adjust the Gamma Curve ------------------------------------------------*/
LCD_WriteReg(R48, 0x0006);
LCD_WriteReg(R49, 0x0101);
LCD_WriteReg(R50, 0x0003);
LCD_WriteReg(R53, 0x0106);
LCD_WriteReg(R54, 0x0b02);
LCD_WriteReg(R55, 0x0302);
LCD_WriteReg(R56, 0x0707);
LCD_WriteReg(R57, 0x0007);
LCD_WriteReg(R60, 0x0600);
LCD_WriteReg(R61, 0x020b);

/* Set GRAM area ---------------------------------------------------------*/
LCD_WriteReg(R80, 0x0000); /* Horizontal GRAM Start Address */
LCD_WriteReg(R81, 0x00EF); /* Horizontal GRAM End Address */
LCD_WriteReg(R82, 0x0000); /* Vertical GRAM Start Address */
LCD_WriteReg(R83, 0x013F); /* Vertical GRAM End Address */
LCD_WriteReg(R96,  0xa700); /* Gate Scan Line */
LCD_WriteReg(R97,  0x0001); /* NDL,VLE, REV */
LCD_WriteReg(R106, 0x0000); /* set scrolling line */
/* Partial Display Control -----------------------------------------------*/
LCD_WriteReg(R128, 0x0000);
LCD_WriteReg(R129, 0x0000);
LCD_WriteReg(R130, 0x0000);
LCD_WriteReg(R131, 0x0000);
LCD_WriteReg(R132, 0x0000);
LCD_WriteReg(R133, 0x0000);
/* Panel Control ---------------------------------------------------------*/
LCD_WriteReg(R144, 0x0010);
LCD_WriteReg(R146, 0x0000);
LCD_WriteReg(R147, 0x0003);
LCD_WriteReg(R149, 0x0110);
LCD_WriteReg(R151, 0x0000);
LCD_WriteReg(R152, 0x0000);
/* Set GRAM write direction and BGR = 1 */
/* I/D=01 (Horizontal : increment, Vertical : decrement) */
/* AM=1 (address is updated in vertical writing direction) */
LCD_WriteReg(R3, 0x1018);
LCD_WriteReg(R7, 0x0173); /* 262K color and display ON */
```

```
}

/**
  * @brief  Sets the cursor position.
  * @param  Xpos: specifies the X position.
  * @param  Ypos: specifies the Y position.
  * @retval None
  */
void LCD_SetCursor(uint8_t Xpos, uint16_t Ypos)
{
    LCD_WriteReg(R32, Xpos);
    LCD_WriteReg(R33, Ypos);
}

void rt_hw_lcd_update(rtgui_rect_t *rect)
{
    /* nothing for none-DMA mode driver */
}

rt_uint8_t * rt_hw_lcd_get_framebuffer(void)
{
    return RT_NULL; /* no framebuffer driver */
}

void rt_hw_lcd_set_pixel(rtgui_color_t *c, rt_base_t x, rt_base_t y)
{
    unsigned short p;

    /* get color pixel */
    p = rtgui_color_to_565p(*c);

    /* set x and y */
    LCD_SetCursor(y, 319 - x);
    LCD_WriteRAMWord(p);
}

void rt_hw_lcd_get_pixel(rtgui_color_t *c, rt_base_t x, rt_base_t y)
{
    /* set x and y */
    LCD_SetCursor(y, 319 - x);

    *c = rtgui_color_from_565p(0xffff);
}

void rt_hw_lcd_draw_hline(rtgui_color_t *c, rt_base_t x1, rt_base_t x2, rt_base_t y)
{
    unsigned short p;

    /* get color pixel */
    p = rtgui_color_to_565p(*c);

    LCD_SetCursor(y, 319 - x1);
    LCD_WriteRAM_Prepare(); /* Prepare to write GRAM */
```

```
    while (x1 < x2)
    {
        LCD_WriteRAM(p);
        x1 ++;
    }
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
}

void rt_hw_lcd_draw_vline(rtgui_color_t *c, rt_base_t x, rt_base_t y1, rt_base_t y2)
{
    unsigned short p;

    /* get color pixel */
    p = rtgui_color_to_565p(*c);

    LCD_SetCursor(y1, 319 - x);
    while (y1 < y2)
    {
        LCD_WriteRAMWord(p);

        y1++;
        LCD_SetCursor(y1, 319 - x);
    }
}

void rt_hw_lcd_draw_raw_hline(rt_uint8_t *pixels, rt_base_t x1, rt_base_t x2, rt_base_t y)
{
    rt_uint16_t *ptr;

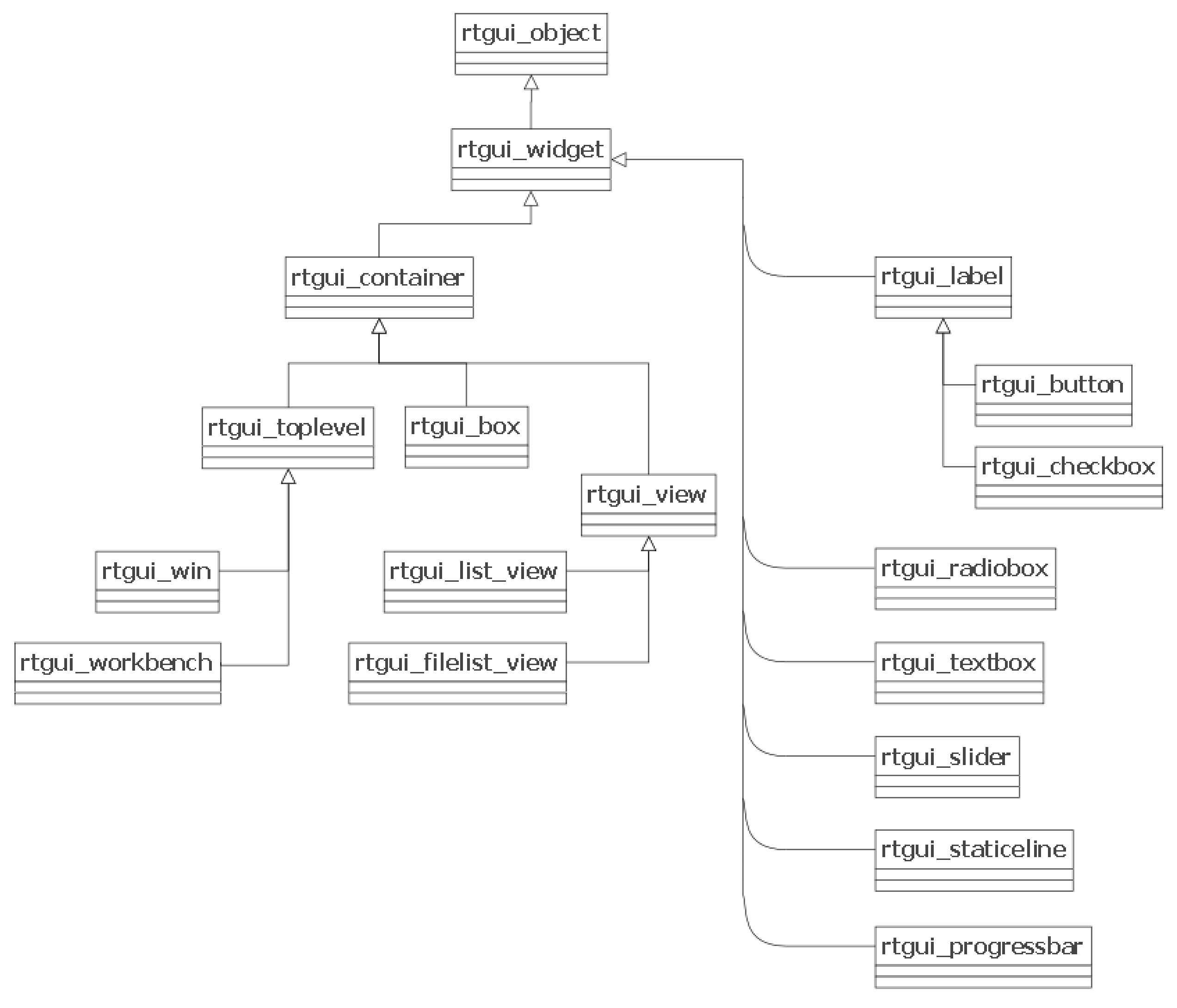
    /* get pixel */
    ptr = (rt_uint16_t*) pixels;

    LCD_SetCursor(y, 319 - x1);
    LCD_WriteRAM_Prepare(); /* Prepare to write GRAM */
    while (x1 < x2)
    {
        LCD_WriteRAM(*ptr);
        x1 ++; ptr ++;
    }
    LCD_CtrlLinesWrite(LCD_NCS_GPIO_PORT, LCD_NCS_PIN, Bit_SET);
}

rt_err_t rt_hw_lcd_init(void)
{
    LCD_Setup();

    /* add lcd driver into graphic driver */
    rtgui_graphic_driver_add(&_rtgui_lcd_driver);
    return RT_EOK;
}
```

15.10 编程说明



上图是RT-Thread/GUI控件实现的类继承图。rtgui_object是GUI系统中的根类。而rtgui_widget是GUI控件中的根类。

rtgui_widget	根控件类
rtgui_label	标签类，用于显示文本标签
rtgui_button	按钮类，用于接受用户输入以触发相应动作
rtgui_checkbox	checkbox类，用于二值选择/二值状态显示
rtgui_radiobox	radiobox类，用于多值选择/多值状态显示
rtgui_text	文本框类，用于接受用户文本输入
rtgui_slider	slider类
rtgui_staticline	静态(立体)线条类
rtgui_progressbar	进度条类，用于显示过程进度情况
rtgui_container	控件容器类
rtgui_toplevel	顶层控件类，服务端事件发送事件消息过来的最顶层接收端。
rtgui_win	窗口类
rtgui_workbench	workbench类
rtgui_box	box自动布局引擎类，按参数可细分为水平布局和垂直布局
rtgui_view	视图类
rtgui_list_view	列表视图类
rtgui_filelist_view	目录文件列表

15.10.1 widget控件

rtgui_widget是RT-Thread/GUI系统中控件相关的基类，描述了GUI中一个控件的基本特性，定义了最基本的事件处理方法。在GUI中的其他各种控件都是从它为源头进行派生，所有通过它派生的对象都能够通过RTGUI_WIDGET(obj)的宏转换成rtgui_widget控件来使用，并调用相应的函数方法进行处理。

控件由几部分组成，首先是它的位置信息：widget->extent。另外，它作为一个独立的基本元素，保留了自己的前景色、背景色，文字对齐模式以及所用的字体。前景色、背景色，文字对齐模式及字体，在RT-Thread/GUI里统称为GC，即图形上下文。对于控件对象，可以通过如下宏获得相应的数据：(w都是一个rtgui_widget对象)

- RTGUI_WIDGET_FOREGROUND(w)
- RTGUI_WIDGET_BACKGROUND(w)
- RTGUI_WIDGET_TEXTALIGN(w)
- RTGUI_WIDGET_FONT(w)

内部则由如下结构体所定义：

```
struct rtgui_gc
{
    /* 前景色和背景色 */
    rtgui_color_t foreground, background;

    /* 文本对齐模式 */
    rt_base_t textalign;

    /* 字体 */
    rtgui_font_t* font;
};
```

文本对齐模式当前支持：

```
enum RTGUI_ALIGN
{
    RTGUI_ALIGN_NOT                 = 0x00,
    RTGUI_ALIGN_CENTER_HORIZONTAL   = 0x01,           /* 水平居中 */
    RTGUI_ALIGN_LEFT                = RTGUI_ALIGN_NOT, /* 左对齐   */
    RTGUI_ALIGN_TOP                 = RTGUI_ALIGN_NOT, /* 顶部对齐 */
    RTGUI_ALIGN_RIGHT               = 0x02,            /* 右对齐   */
    RTGUI_ALIGN_BOTTOM              = 0x04,            /* 底部对齐 */
    RTGUI_ALIGN_CENTER_VERTICAL     = 0x08,           /* 垂直居中 */
    RTGUI_ALIGN_EXPAND              = 0x10,            /* 自动扩展 */
    RTGUI_ALIGN_STRETCH             = 0x20             /* 自动延伸 */
};
```

其中，RTGUI_ALIGN_EXPAND和RTGUI_ALIGN_STRETCH仅在rtgui_box实现控件位置自动布局时才有作用。

`rtgui_widget_t *` **`rtgui_widget_create`**(*rtgui_type_t *widget_type*)

`void` **`rtgui_widget_destroy`**(*rtgui_widget_t* widget*)

这两个函数分别用于创建或删除一个控件。创建时需要指定控件的类型，如果创建成功，返回相应的控件对象指针。如果创建失败，返回RT_NULL。删除控件时，需要给出rtgui_widget_create返回的控件指针，destroy函数会根据控件的析构函数进行内存释放操作。

`void` **`rtgui_widget_set_event_handler`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`rt_bool_t` **`rtgui_widget_event_handler`**(*rtgui_widget_t* widget, rtgui_event_t* event*)

这两个函数分别用于设置或获得控件的事件处理函数指针。其中事件处理函数的声明形式是：

```
typedef rt_bool_t (*rtgui_event_handler_ptr)(struct rtgui_widget* widget,
    struct rtgui_event* event);
```

参数widget指向当前获得事件的控件；参数event指向当前处理的事件。如果事件处理成功(即此事件是控件所感兴趣的事件，被这个空间所处理)，那么应该返回RT_TRUE，上层事件处理函数将不再把此事件派发给其他事件处理函数进行处理。如果返回RT_FALSE，上层事件处理函数将继续把这个事件传递给其他控件进行解析。

`void` **`rtgui_widget_focus`**(*rtgui_widget_t * widget*)

`void` **`rtgui_widget_unfocus`**(*rtgui_widget_t *widget*)

这两个函数用于设置控件的焦点或去焦点。在去焦点的时候，控件的on_focus_out虚函数将被自动调用（同时调用的还是控件的刷新动作）。

`void` **`rtgui_widget_set_onfocus`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_onunfocus`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_ondraw`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_onmouseclick`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_onkey`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_onsize`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

`void` **`rtgui_widget_set_oncommand`**(*rtgui_widget_t* widget, rtgui_event_handler_ptr handler*)

上面的几个函数分别设置控件的几个虚拟(回调)函数指针。在小模式版本中，后面几个函数将不支持，它们仅仅出现在标准版本中。回调函数的声明形式和事件处理函数的声明形式相同(其中，event参数在大多数情况下会被设置成RT_NULL)。

void **rtgui_widget_get_rect**(*rtgui_widget_t* widget, rtgui_rect_t *rect*)
: 这个函数用于获得或设置控件的矩形信息。

  注：返回的rect信息是控件的相对位置，所以实际上仅仅返回控件矩形的宽度和高度(因为rect->x1和rect->y1都等于0)。

void **rtgui_widget_set_rect**(*rtgui_widget_t* widget, rtgui_rect_t* rect*)
: 这个函数用于设置控件的矩形信息。注：此处指定的rect信息是控件的绝对位置。

void **rtgui_widget_point_to_device**(*rtgui_widget_t * widget, rtgui_point_t * point*)
void **rtgui_widget_rect_to_device**(*rtgui_widget_t * widget, rtgui_rect_t * rect*)
: 这两个函数分别实现了相对点和相对矩形到设备的绝对化转换，变成绝对的点和矩形区域(即相对于绘图硬件设备)。

void **rtgui_widget_point_to_logic**(*rtgui_widget_t* widget, rtgui_point_t * point*)
void **rtgui_widget_rect_to_logic**(*rtgui_widget_t* widget, rtgui_rect_t* rect*)
: 这两个函数分别实现了绝对点和绝对矩形到设备的相对化转换，变成相对的点和矩形区域(即相对于控件)。这两个函数上面两个的逆操作。

void **rtgui_widget_move_to_logic**(*rtgui_widget_t* widget, int dx, int dy*)
: 这个函数实现了控件位置的相对移动。

void **rtgui_widget_update_clip**(*rtgui_widget_t* widget*)
: 这个函数实现了控件的剪切域更新。

rtgui_widget_t* **rtgui_widget_get_toplevel**(*rtgui_widget_t* widget*)
: 这个函数用于获取控件所属的最顶层祖先控件。

void **rtgui_widget_show**(*rtgui_widget_t* widget*)
void **rtgui_widget_hide**(*rtgui_widget_t* widget*)
: 这两个函数用于显示/隐藏控件。

void **rtgui_widget_update**(*rtgui_widget_t* widget*)
: 这个函数用于更新控件绘图，即实现控件的重绘。

rtgui_color_t **rtgui_widget_get_parent_foreground**(*rtgui_widget_t* widget*)
: 这个函数用于获得控件的上层可见父控件的前景色。

rtgui_color_t **rtgui_widget_get_parent_background**(*rtgui_widget_t* widget*)
: 这个函数用于获得控件的上层可见父控件的背景色。

rtgui_widget_t* **rtgui_widget_get_next_sibling**(*rtgui_widget_t* widget*)
: 这个函数用于获得控件的下一个兄弟控件。

rtgui_widget_t* **rtgui_widget_get_prev_sibling**(*rtgui_widget_t* widget*)
: 这个函数用于获得控件的前一个兄弟控件。

rtgui_widget看起来更像一个抽象的控件，包含了众多公共的方法、属性，而最终的显示部分则留给了从它派生出来的子对象。在本章的末尾:_gui_mywidget 有一个如何从它派生出自己控件的实例，可以参考。

15.10.2 container控件

rtgui_container被设计成RTGUI中的一个容器类，它允许在它之下包含数个子控件（子控件的大小位置必须在container大小位置之内，否则自动被剪切掉）。rtgui_container内的控件绘图次序是按照从上到下的次序进行的。

container控件的结构定义为：

```
struct rtgui_container
{
    /* 继承自rtgui_widget控件 */
    struct rtgui_widget parent;

    /* 这个container控件中当前获得焦点的控件 */
    struct rtgui_widget* focused;

    /* container控件下的子控件列表 */
    rtgui_list_t children;
};
```

void **rtgui_container_add_child**(*rtgui_container_t *container, rtgui_widget_t* child*)
这个函数用于在rtgui_container容器中添加一个子控件。

void **rtgui_container_remove_child**(*rtgui_container_t *container, rtgui_widget_t* child*)
这个函数用于在rtgui_container容器中删除一个子控件。

void **rtgui_container_destroy_children**(*rtgui_container_t *container*)
这个函数用于删除容器中所有的子控件(并对每个子控件做析构)。

rtgui_widget_t* **rtgui_container_get_first_child**(*rtgui_container_t* container*)
这个函数用于获得容器中第一个子控件。

rt_bool_t **rtgui_container_event_handler**(*rtgui_widget_t* widget, rtgui_event_t* event*)
这个函数是rtgui_container容器类的默认事件处理函数。从rtgui_container派生的子类可以调用这个函数获得父类的事件处理方法。

rt_bool_t **rtgui_container_dispatch_event**(*rtgui_container_t *container, rtgui_event_t* event*)
这个函数用于向子控件派发事件。

rt_bool_t **rtgui_container_dispatch_mouse_event**(*rtgui_container_t *container, struct rtgui_event_mouse* event*)
这个函数用于向子控件派发鼠标事件。鼠标事件会根据子控件的位置信息选择是否进行派发，如果鼠标坐标位置在子控件之外，将不派发鼠标事件到这个子控件中。

15.10.3 label控件

label控件是一个文本标签，在屏幕相应位置显示相应的文本。其中最主要的属性是显示的文字文本。

rtgui_label_t* **rtgui_label_create**(*const char* text*)
这个函数用于创建rtgui_label控件。

注：标签的文本应该还包括一些其他特性，例如字体，文本显示对齐方式等。这些特性都可以把rtgui_label对象转换成rtgui_widget对象以应用rtgui_widget控件的方法。

void **rtgui_label_destroy**(*rtgui_label_t* label*)
这个函数用于删除rtgui_label控件。

rt_bool_t **rtgui_label_event_handler**(*struct rtgui_widget* widget, struct rtgui_event* event*)
这个是rtgui_label控件的默认事件处理函数。

void **rtgui_label_set_text**(*rtgui_label_t* label, const char* text*)
    这个函数用于设置标签的文本。

char* **rtgui_label_get_text**(*rtgui_label_t* label*)
    这两个函数用于获得标签的文本。

rtgui_label的例子：

```
/*
 * 程序清单：label控件演示
 *
 * 这个例子会在创建出的view上添加几个不同类型的label控件
 */
#include "demo_view.h"
#include <rtgui/widgets/label.h>

/* 创建用于演示label控件的视图 */
rtgui_view_t* demo_view_label(rtgui_workbench_t* workbench)
{
	rtgui_rect_t rect;
	rtgui_view_t* view;
	rtgui_label_t* label;
	rtgui_font_t* font;

	/* 先创建一个演示用的视图 */
	view = demo_view(workbench, "Label View");

	/* 获得视图的位置信息 */
	demo_view_get_rect(view, &rect);
	rect.x1 += 5;
	rect.x2 -= 5;
	rect.y1 += 5;
	rect.y2 = rect.y1 + 20;
	/* 创建一个label控件 */
	label = rtgui_label_create("Red Left");
	/* 设置label控件上的文本对齐方式为：左对齐 */
	RTGUI_WIDGET_TEXTALIGN(RTGUI_WIDGET(label)) = RTGUI_ALIGN_LEFT;
	/* 设置label控件的前景色为红色 */
	RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(label)) = red;
	/* 设置label的位置 */
	rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
	/* view是一个container控件，调用add_child方法添加这个label控件 */
	rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));

	/* 获得视图的位置信息 */
	demo_view_get_rect(view, &rect);
	rect.x1 += 5;
	rect.x2 -= 5;
	rect.y1 += 5 + 25;
	rect.y2 = rect.y1 + 20;
	/* 创建一个label控件 */
	label = rtgui_label_create("Blue Right");
	/* 设置label控件上的文本对齐方式为：右对齐 */
	RTGUI_WIDGET_TEXTALIGN(RTGUI_WIDGET(label)) = RTGUI_ALIGN_RIGHT;
```

```
47          /* 设置label控件的前景色为蓝色 */
48          RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(label)) = blue;
49          /* 设置label的位置 */
50          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
51          /* view是一个container控件，调用add_child方法添加这个label控件 */
52          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
53
54          /* 获得视图的位置信息 */
55          demo_view_get_rect(view, &rect);
56          rect.x1 += 5;
57          rect.x2 -= 5;
58          rect.y1 += 5 + 25 + 25;
59          rect.y2 = rect.y1 + 20;
60          /* 创建一个label控件 */
61          label = rtgui_label_create("Green Center");
62          /* 设置label控件的前景色为绿色 */
63          RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(label)) = green;
64          /* 设置label控件上的文本对齐方式为：右对齐 */
65          RTGUI_WIDGET_TEXTALIGN(RTGUI_WIDGET(label)) = RTGUI_ALIGN_CENTER_HORIZONTAL;
66          /* 设置label的位置 */
67          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
68          /* view是一个container控件，调用add_child方法添加这个label控件 */
69          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
70
71          /* 获得视图的位置信息 */
72          demo_view_get_rect(view, &rect);
73          rect.x1 += 5;
74          rect.x2 -= 5;
75          rect.y1 += 5 + 25 + 25 + 25;
76          rect.y2 = rect.y1 + 20;
77          /* 创建一个label控件 */
78          label = rtgui_label_create("12 font");
79          /* 设置字体为12点阵的asc字体 */
80          font = rtgui_font_refer("asc", 12);
81          RTGUI_WIDGET_FONT(RTGUI_WIDGET(label)) = font;
82          /* 设置label的位置 */
83          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
84          /* view是一个container控件，调用add_child方法添加这个label控件 */
85          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
86
87          /* 获得视图的位置信息 */
88          demo_view_get_rect(view, &rect);
89          rect.x1 += 5;
90          rect.y1 += 5 + 25 + 25 + 25 + 25;
91          rect.y2 = rect.y1 + 20;
92          /* 创建一个label控件 */
93          label = rtgui_label_create("16 font");
94          /* 设置字体为16点阵的asc字体 */
95          font = rtgui_font_refer("asc", 16);
96          RTGUI_WIDGET_FONT(RTGUI_WIDGET(label)) = font;
97          /* 设置label的位置 */
98          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
99          /* view是一个container控件，调用add_child方法添加这个label控件 */
```

```
100         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
101 
102         return view;
103 }
```

15.10.4 button控件

button控件是一个带标签的按钮控件，继承自rtgui_label，所以label的方法也可应用于button控件。

`rtgui_button_t*` **`rtgui_button_create`**(*char* text*)
: 这个函数用于创建一个按钮控件。

`rtgui_button_t*` **`rtgui_pushbutton_create`**(*char* text*)
: 这个函数用于创建push按钮，即按下按钮时，按钮的状态将被保持，而不是当鼠标抬起时，按钮自动恢复为原样。只有当鼠标再次点击按钮时，它才能恢复到初始状态。

`void` **`rtgui_button_destroy`**(*rtgui_button_t* btn*)
: 这个函数用于删除按钮控件。

`void` **`rtgui_button_set_pressed_image`**(*rtgui_button_t* btn, rtgui_image_t* image*)
: 这个函数用于设置按钮的按下时显示的图像。image是一个rtgui_image对象，具体参见rtgui_image类的解释。

`void` **`rtgui_button_set_unpressed_image`**(*rtgui_button_t* btn, rtgui_image_t* image*)
: 这个函数用于设置按钮正常状态时显示的图像。image是一个rtgui_image对象，具体参见rtgui_image类的解释。

`void` **`rtgui_button_set_onbutton`**(*rtgui_button_t* btn, rtgui_onbutton_func_t func*)
: 这个函数用于设置按钮控件的按下事件时，被调用的虚拟(回调)函数指针。注：对于push按钮，每次在按钮上点击按钮都会触发相应的onbutton函数回调。

`rt_bool_t` **`rtgui_button_event_handler`**(*struct rtgui_widget* widget, struct rtgui_event* event*)
: rtgui_button类的默认事件处理函数。

rtgui_button的例子：

```
1  /*
2   * 程序清单：button控件演示
3   *
4   * 这个例子会在创建出的view上添加几个不同类型的button控件
5   */
6  
7  #include "demo_view.h"
8  #include <rtgui/widgets/button.h>
9  
10 /* 创建用于演示button控件的视图 */
11 rtgui_view_t* demo_view_button(rtgui_workbench_t* workbench)
12 {
13         rtgui_rect_t rect;
14         rtgui_view_t* view;
15         rtgui_button_t* button;
16         rtgui_font_t* font;
17 
```

```
18      /* 先创建一个演示用的视图 */
19      view = demo_view(workbench, "Button View");
20
21      /* 获得视图的位置信息 */
22      demo_view_get_rect(view, &rect);
23      rect.x1 += 5;
24      rect.x2 = rect.x1 + 100;
25      rect.y1 += 5;
26      rect.y2 = rect.y1 + 20;
27      /* 创建一个button控件 */
28      button = rtgui_button_create("Red");
29      /* 设置label控件的前景色为红色 */
30      RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(button)) = red;
31      /* 设置button的位置 */
32      rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
33      /* view是一个container控件，调用add_child方法添加这个button控件 */
34      rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
35
36      /* 获得视图的位置信息 */
37      demo_view_get_rect(view, &rect);
38      rect.x1 += 5;
39      rect.x2 = rect.x1 + 100;
40      rect.y1 += 5 + 25;
41      rect.y2 = rect.y1 + 20;
42      /* 创建一个button控件 */
43      button = rtgui_button_create("Blue");
44      /* 设置label控件的前景色为蓝色 */
45      RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(button)) = blue;
46      /* 设置button的位置 */
47      rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
48      /* view是一个container控件，调用add_child方法添加这个button控件 */
49      rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
50
51      /* 获得视图的位置信息 */
52      demo_view_get_rect(view, &rect);
53      rect.x1 += 5;
54      rect.x2 = rect.x1 + 100;
55      rect.y1 += 5 + 25 + 25;
56      rect.y2 = rect.y1 + 20;
57      /* 创建一个button控件 */
58      button = rtgui_button_create("12 font");
59      /* 设置字体为12点阵的asc字体 */
60      font = rtgui_font_refer("asc", 12);
61      RTGUI_WIDGET_FONT(RTGUI_WIDGET(button)) = font;
62      /* 设置button的位置 */
63      rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
64      /* view是一个container控件，调用add_child方法添加这个button控件 */
65      rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
66
67      /* 获得视图的位置信息 */
68      demo_view_get_rect(view, &rect);
69      rect.x1 += 5;
70      rect.x2 = rect.x1 + 100;
```

```
        rect.y1 += 5 + 25 + 25 + 25;
        rect.y2 = rect.y1 + 20;
        /* 创建一个button控件 */
        button = rtgui_button_create("16 font");
        /* 设置字体为16点阵的asc字体 */
        font = rtgui_font_refer("asc", 16);
        RTGUI_WIDGET_FONT(RTGUI_WIDGET(button)) = font;
        /* 设置button的位置 */
        rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
        /* view是一个container控件，调用add_child方法添加这个button控件 */
        rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));

        return view;
}
```

15.10.5 textbox控件

struct rtgui_textbox* **rtgui_textbox_create**(*const char* text, rt_uint8_t flag*)

这个函数用于创建rtgui_textbox控件。创建时，参数text指定文本框中的文本；参数flag指定文本框的风格，值可以包括：

```
#define RTGUI_TEXTBOX_SINGLE            0x00 /* 单行文本框(默认) */
#define RTGUI_TEXTBOX_MULTI             0x01 /* 多行文本框(当前版本不支持) */
#define RTGUI_TEXTBOX_MASK              0x02 /* 文本框中的文本显示成'*' */
```

void **rtgui_textbox_destroy**(*struct rtgui_textbox* box*)

这个函数用于删除rtgui_textbox控件。

rt_bool_t **rtgui_textbox_event_handler**(*struct rtgui_widget* widget, struct rtgui_event* event*)

这个函数是文本框控件默认的事件处理函数。

void **rtgui_textbox_set_value**(*struct rtgui_textbox* box, const char* text*)

这个函数用于向文本框控件设置文本。

const char* **rtgui_textbox_get_value**(*struct rtgui_textbox* box*)

这个函数用于获得文本框控件的文本。

void **rtgui_widget_set_line_length**(*struct rtgui_textbox* box, rt_size_t length*)

这个函数用于设置文本框中的文本字符串空间。

rtgui_textbox的例子：

```
/*
 * 程序清单：texbox控件演示
 *
 * 这个例子会在创建出的view上添加几个不同类型的textbox控件
 */
#include "demo_view.h"
#include <rtgui/widgets/label.h>
#include <rtgui/widgets/textbox.h>

/* 创建用于演示textbox控件的视图 */
rtgui_view_t* demo_view_textbox(rtgui_workbench_t* workbench)
```

```
12  {
13          rtgui_rect_t rect, textbox_rect;
14          rtgui_view_t* view;
15          rtgui_label_t* label;
16          rtgui_textbox_t* text;
17
18          /* 先创建一个演示用的视图 */
19          view = demo_view(workbench, "TextBox View");
20
21          /* 获得视图的位置信息 */
22          demo_view_get_rect(view, &rect);
23          rect.x1 += 5;
24          rect.x2 = rect.x1 + 30;
25          rect.y1 += 5;
26          rect.y2 = rect.y1 + 20;
27          /* 创建一个label控件 */
28          label = rtgui_label_create("名字: ");
29          /* 设置label的位置 */
30          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
31          /* view是一个container控件，调用add_child方法添加这个label控件 */
32          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
33
34          /* 让textbox_rect赋值到rect，以计算textbox控件的位置 */
35          textbox_rect = rect;
36          textbox_rect.x1 = textbox_rect.x2 + 5;
37          textbox_rect.x2 = textbox_rect.x1 + 160;
38          /* 创建一个textbox控件 */
39          text = rtgui_textbox_create("bernard",RTGUI_TEXTBOX_SINGLE);
40          /* 设置textbox控件的位置 */
41          rtgui_widget_set_rect(RTGUI_WIDGET(text), &textbox_rect);
42          /* 添加textbox控件到视图中 */
43          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(text));
44
45          /* 计算下一个label控件的位置 */
46          rect.y1 += 23;
47          rect.y2 = rect.y1 + 20;
48          /* 创建一个label控件 */
49          label = rtgui_label_create("邮件: ");
50          /* 设置label的位置 */
51          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
52          /* 添加label控件到视图中 */
53          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
54          textbox_rect = rect;
55          textbox_rect.x1 = textbox_rect.x2 + 5;
56          textbox_rect.x2 = textbox_rect.x1 + 160;
57          /* 创建一个textbox控件 */
58          text = rtgui_textbox_create("bernard.xiong@gmail.com",RTGUI_TEXTBOX_SINGLE);
59          /* 设置textbox控件的位置 */
60          rtgui_widget_set_rect(RTGUI_WIDGET(text), &textbox_rect);
61          /* 添加textbox控件到视图中 */
62          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(text));
63
64          rect.y1 += 23;
```

```
65          rect.y2 = rect.y1 + 20;
66          /* 创建一个label控件 */
67          label = rtgui_label_create("密码: ");
68          /* 设置label的位置 */
69          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
70          /* 添加label控件到视图中 */
71          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
72          textbox_rect = rect;
73          textbox_rect.x1 = textbox_rect.x2 + 5;
74          textbox_rect.x2 = textbox_rect.x1 + 160;
75          /* 创建一个textbox控件 */
76          text = rtgui_textbox_create("rt-thread",RTGUI_TEXTBOX_SINGLE);
77          /* 设置textbox显示文本为掩码形式(即显示为*号，适合于显示密码的情况） */
78          text->flag |= RTGUI_TEXTBOX_MASK;
79          /* 设置textbox控件的位置 */
80          rtgui_widget_set_rect(RTGUI_WIDGET(text), &textbox_rect);
81          /* 添加textbox控件到视图中 */
82          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(text));
83
84          rect.y1 += 23;
85          rect.y2 = rect.y1 + 20;
86          /* 创建一个label控件 */
87          label = rtgui_label_create("主页: ");
88          /* 设置label的位置 */
89          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
90          /* 添加label控件到视图中 */
91          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
92          textbox_rect = rect;
93          textbox_rect.x1 = textbox_rect.x2 + 5;
94          textbox_rect.x2 = textbox_rect.x1 + 160;
95          /* 创建一个textbox控件 */
96          text = rtgui_textbox_create("http://www.rt-thread.org",RTGUI_TEXTBOX_SINGLE);
97          /* 设置textbox控件的位置 */
98          rtgui_widget_set_rect(RTGUI_WIDGET(text), &textbox_rect);
99          /* 添加textbox控件到视图中 */
100         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(text));
101
102         return view;
103 }
```

15.10.6 checkbox控件

checkbox控件可以看成是一个能够显示二进制状态的文本标签，并能够根据用户的输入来更改状态。checkbox控件也是继承自label控件，label控件的方法也适合于checkbox控件。

struct rtgui_checkbox* **rtgui_checkbox_create**(*unsigned char* text, rt_bool_t checked*)

  这个函数用于创建一个checkbox控件。参数text指示出checkbox的文本标签，参数checked为RT_TRUE时，创建的checkbox控件默认是checked状态。

void **rtgui_checkbox_destroy**(*rtgui_checkbox_t* checkbox*)

  这个函数用于删除一个checkbox控件。

void **rtgui_checkbox_set_checked**(*rtgui_checkbox_t* checkbox, rt_bool_t checked*)

`rt_bool_t` **`rtgui_checkbox_get_checked`**(*rtgui_checkbox_t* checkbox*)
　　这个函数用于设置或获取checkbox控件的checked状态。

`rt_bool_t` **`rtgui_checkbox_event_handler`**(*struct rtgui_widget* widget, struct rtgui_event* event*)
　　这个函数是checkbox控件默认的事件处理函数。

rtgui_checkbox的例子：

```
/*
 * 程序清单：checkbox控件演示
 *
 * 这个例子会在创建出的view上添加几个checkbox控件
 */

#include "demo_view.h"
#include <rtgui/widgets/checkbox.h>

/* 创建用于演示checkbox控件的视图 */
rtgui_view_t* demo_view_checkbox(rtgui_workbench_t* workbench)
{
	rtgui_rect_t rect;
	rtgui_view_t* view;
	rtgui_checkbox_t* checkbox;
	rtgui_font_t* font;

	/* 先创建一个演示用的视图 */
	view = demo_view(workbench, "CheckBox View");

	/* 获得视图的位置信息 */
	demo_view_get_rect(view, &rect);
	rect.x1 += 5;
	rect.x2 = rect.x1 + 100;
	rect.y1 += 5;
	rect.y2 = rect.y1 + 20;
	/* 创建一个checkbox控件 */
	checkbox = rtgui_checkbox_create("Red",RT_TRUE);
	/* 设置前景色为红色 */
	RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(checkbox)) = red;
	/* 设置checkbox的位置 */
	rtgui_widget_set_rect(RTGUI_WIDGET(checkbox), &rect);
	/* view是一个container控件，调用add_child方法添加这个checkbox控件 */
	rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(checkbox));

	/* 获得视图的位置信息 */
	demo_view_get_rect(view, &rect);
	rect.x1 += 5;
	rect.x2 = rect.x1 + 100;
	rect.y1 += 5 + 25;
	rect.y2 = rect.y1 + 20;
	/* 创建一个checkbox控件 */
	checkbox = rtgui_checkbox_create("Blue",RT_TRUE);
	/* 设置前景色为蓝色 */
	RTGUI_WIDGET_FOREGROUND(RTGUI_WIDGET(checkbox)) = blue;
	/* 设置checkbox的位置 */
```

```
47          rtgui_widget_set_rect(RTGUI_WIDGET(checkbox), &rect);
48          /* view是一个container控件，调用add_child方法添加这个checkbox控件 */
49          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(checkbox));
50
51          /* 获得视图的位置信息 */
52          demo_view_get_rect(view, &rect);
53          rect.x1 += 5;
54          rect.x2 = rect.x1 + 100;
55          rect.y1 += 5 + 25 + 25;
56          rect.y2 = rect.y1 + 20;
57          /* 创建一个checkbox控件 */
58          checkbox = rtgui_checkbox_create("12 font",RT_TRUE);
59          /* 设置字体为12点阵 */
60          font = rtgui_font_refer("asc", 12);
61          RTGUI_WIDGET_FONT(RTGUI_WIDGET(checkbox)) = font;
62          /* 设置checkbox的位置 */
63          rtgui_widget_set_rect(RTGUI_WIDGET(checkbox), &rect);
64          /* view是一个container控件，调用add_child方法添加这个checkbox控件 */
65          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(checkbox));
66
67          /* 获得视图的位置信息 */
68          demo_view_get_rect(view, &rect);
69          rect.x1 += 5;
70          rect.x2 = rect.x1 + 100;
71          rect.y1 += 5 + 25 + 25 + 25;
72          rect.y2 = rect.y1 + 20;
73          /* 创建一个checkbox控件 */
74          checkbox = rtgui_checkbox_create("16 font",RT_TRUE);
75          /* 设置字体为16点阵 */
76          font = rtgui_font_refer("asc", 16);
77          RTGUI_WIDGET_FONT(RTGUI_WIDGET(checkbox)) = font;
78          /* 设置checkbox的位置 */
79          rtgui_widget_set_rect(RTGUI_WIDGET(checkbox), &rect);
80          /* view是一个container控件，调用add_child方法添加这个checkbox控件 */
81          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(checkbox));
82
83          return view;
84  }
```

15.10.7 radiobox控件

`struct rtgui_radiobox*` **`rtgui_radiobox_create`**(*const char* label, int orient, char** items, int number*)

这个函数用于创建一个radiobox控件。参数label指示出radiobox控件的文本标签，参数items指示出radiobox控件中包含的各个选择项的文本，参数number指示出总共存在的文本项。

`void` **`rtgui_radiobox_destroy`**(*struct rtgui_radiobox* radiobox*)

这个函数用于删除一个radiobox控件。

`void` **`rtgui_radiobox_set_selection`**(*struct rtgui_radiobox* radiobox, int selection*)

这个函数用于设置radiobox控件中当前选择的选择项。参数selection指示出选择项的序号。

`int` **`rtgui_radiobox_get_selection`**(*struct rtgui_radiobox* radiobox*)
这两个函数用于获取radiobox控件中当前选择的选择项。

`rt_bool_t` **`rtgui_radiobox_event_handler`**(*struct rtgui_widget* widget, struct rtgui_event* event*)
这个函数是radiobox控件默认的事件处理函数。

`void` **`rtgui_radiobox_set_orientation`**(*struct rtgui_radiobox* radiobox, int orientation*)
这个函数用于设置radiobox控件显示选择项的方向，参数orientation指示出具体的方向，可以设置的值：

- RTGUI_HORIZONTAL - 水平方向显示
- RTGUI_VERTICAL - 垂直方向显示

rtgui_radiobox的例子：

```
/*
 * 程序清单：radiobox控件演示
 *
 * 这个例子会在创建出的view上添加两个不同方向的radiobox控件
 */

#include "demo_view.h"
#include <rtgui/widgets/radiobox.h>

/* 用于显示垂直方向的radio文本项数组 */
static char* radio_item_v[5] =
	{
		"one",
		"two",
		"three",
		"item 1",
		"item 2"
	};

/* 用于显示水平方向的radio文本项数组 */
static char* radio_item_h[3] =
	{
		"one", "two", "three"
	};

/* 创建用于演示radiobox控件的视图 */
rtgui_view_t* demo_view_radiobox(rtgui_workbench_t* workbench)
{
	rtgui_rect_t rect;
	rtgui_view_t* view;
	rtgui_radiobox_t* radiobox;

	/* 先创建一个演示用的视图 */
	view = demo_view(workbench, "RadioBox View");

	/* 获得视图的位置信息 */
	demo_view_get_rect(view, &rect);
	rect.x1 += 5;
	rect.x2 -= 5;
```

```
40         rect.y1 += 5;
41         rect.y2 = rect.y1 + 5 * 25;
42 
43         /* 创建一个垂直方向显示的radiobox控件，文本项是radio_item_v数组，共5个项 */
44         radiobox = rtgui_radiobox_create("Radio Box", RTGUI_VERTICAL, radio_item_v, 5);
45         /* 设置当前选择的数组是第0项 */
46         rtgui_radiobox_set_selection(radiobox, 0);
47         /* 添加radiobox控件到视图中 */
48         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(radiobox));
49         /* 设置radiobox控件的位置信息 */
50         rtgui_widget_set_rect(RTGUI_WIDGET(radiobox), &rect);
51 
52         /* 获得视图的位置信息 */
53         demo_view_get_rect(view, &rect);
54         rect.x1 += 5;
55         rect.x2 -= 5;
56         rect.y1 += 5 + 5 * 25;
57         rect.y2 = rect.y1 + 60;
58 
59         /* 创建一个水平方向显示的radiobox控件，文本项是radio_item_h数组，共3个项 */
60         radiobox = rtgui_radiobox_create("Radio Box", RTGUI_HORIZONTAL, radio_item_h, 3);
61         /* 设置当前选择的数组是第0项 */
62         rtgui_radiobox_set_selection(radiobox, 0);
63         /* 添加radiobox控件到视图中 */
64         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(radiobox));
65         /* 设置radiobox控件的位置信息 */
66         rtgui_widget_set_rect(RTGUI_WIDGET(radiobox), &rect);
67 
68         return view;
69 }
```

15.10.8 工作台应用：workbench

在RT-Thread/GUI中，一个workbench也能够看成是一个用户应用，它有自己独立的执行环境，自己独立的事件处理消息队列(这点在设备上下文小节也能够看得出来)。workbench另一个意思是，它是独占一个面板的，在相同的面板上，不可能存在相互交错重叠的两个workbench (一个面板上允许多个workbench存在，但一个时刻只允许一个workbench能够被显示)。

当一个workbench无事可做时，其附着线程将挂起在事件消息队列上，直到等到有新的消息达到时，附着线程才被唤醒并进行消息的处理。

```
1  /* workbench线程的入口函数声明 */
2  extern static void workbench_entry(void* parameter);
3 
4  /* UI应用程序的初始化 */
5  void ui_application_init()
6  {
7      rt_thread_t tid;
8 
9      /* 创建一个线程用于workbench应用 */
10     tid = rt_thread_create("wb",
```

```
        workbench_entry, RT_NULL,
        2048, 25, 10);

    /* 启动线程 */
    if (tid != RT_NULL) rt_thread_startup(tid);
}

/* workbench应用入口 */
static void workbench_entry(void* parameter)
{
    rt_mq_t mq;
    struct rtgui_view* view;
    struct rtgui_workbench* workbench;

    /* 创建相应的事件处理消息队列 */
#ifdef RTGUI_USING_SMALL_SIZE
    mq = rt_mq_create("workbench", 32, 8, RT_IPC_FLAG_FIFO);
#else
    mq = rt_mq_create("workbench", 256, 8, RT_IPC_FLAG_FIFO);
#endif
    /* 注册成为GUI线程 */
    rtgui_thread_register(rt_thread_self(), mq);

    /* 创建workbench */
    workbench = rtgui_workbench_create("main", "workbench");
    if (workbench == RT_NULL) return;

    /* 在workbench创建成功后，可以加入view或window等，此处略 */

    /* 执行workbench的事件循环 */
    rtgui_workbench_event_loop(workbench);

    /* 当从事件循环中退出时，一般代表这个workbench已经关闭 */

    /* 去注册GUI线程 */
    rtgui_thread_deregister(rt_thread_self());
    /* 删除相应的消息队列 */
    rt_mq_delete(mq);
}
```

rtgui_workbench_t * **rtgui_workbench_create**(*const char* panel_name, const unsigned char* title*)

这个函数用于创建一个workbench。在创建时，参数panel_name指定workbench附着的面板名称，参数title指定workbench的标题。如果创建失败，返回RT_NULL。如果创建成功，那么调用'rtgui_widget_get_rect'将能够获得所附着面板的大小。

void **rtgui_workbench_destroy**(*rtgui_workbench_t* workbench*)

这个函数用于删除一个workbench。

rt_bool_t **rtgui_workbench_event_handler**(*rtgui_widget_t* widget, rtgui_event_t* event*)

这个函数是workbench的默认事件处理函数。

void **rtgui_workbench_set_flag**(*rtgui_workbench_t* workbench, rt_uint8_t flag*)

这个函数用于设置workbench的风格，参数flag指定相应的风格参数。当前的风格支持：

- RTGUI_WORKBENCH_FLAG_VISIBLE – workbench是可见的
- RTGUI_WORKBENCH_FLAG_INVISIBLE – workbench是非可见的
- RTGUI_WORKBENCH_FLAG_FULLSCREEN – workbench是全屏的
- RTGUI_WORKBENCH_FLAG_MODAL_MODE – workbench当前处于模式显示中(或者说，当前workbench正在显示一个模式窗口或视图)
- RTGUI_WORKBENCH_FLAG_CLOSEBLE – workbench是可关闭的
- RTGUI_WORKBENCH_FLAG_UNCLOSEBLE – workbench是不可关闭的
- RTGUI_WORKBENCH_FLAG_CLOSED – workbench已经关闭

注：以上这些参数用户最好不要进行设置。

`rt_bool_t` **`rtgui_workbench_event_loop`**(*rtgui_workbench_t* workbench*)
  这个函数是workbench的默认事件处理函数。

`rt_err_t` **`rtgui_workbench_show`**(*rtgui_workbench_t* workbench*)
  这两个函数用于显示一个workbench。

`rt_err_t` **`rtgui_workbench_hide`**(*rtgui_workbench_t* workbench*)
  这两个函数用于隐藏一个workbench。

`void` **`rtgui_workbench_add_view`**(*rtgui_workbench_t* workbench, rtgui_view_t* view*)
  这个函数用于在workbensh上添加一个视图。

`void` **`rtgui_workbench_remove_view`**(*rtgui_workbench_t* workbench, rtgui_view_t* view*)
  这个函数用于在workbensh上删除一个视图。

`void` **`rtgui_workbench_show_view`**(*rtgui_workbench_t* workbench, rtgui_view_t* view*)
  这个函数用于在workbench上显示一个视图(视图必须为非模式视图)。

`void` **`rtgui_workbench_hide_view`**(*rtgui_workbench_t* workbench, rtgui_view_t* view*)
  这个函数用于在workbench上隐藏一个视图(视图必须为非模式视图)。

`rtgui_view_t *` **`rtgui_workbench_get_current_view`**(*rtgui_workbench_t * workbench*)
  这个函数用于获得workbench上当前的视图。

15.10.9 工作台视图：view

如前面说的，视图可以看成是workbench应用上的一个个面，当然一个时刻workbench只能显示其中一个面，这个面可以调用函数 rtgui_workbench_get_current_view 获得。

`rtgui_view_t*` **`rtgui_view_create`**(*const char* title*)
  这个函数用于创建一个视图，参数title指明了视图的标题。注：视图创建处理后，需要加入到一个workbench中才能够使用。

`void` **`rtgui_view_destroy`**(*rtgui_view_t* view*)
  这个函数用于删除一个视图。

`rt_bool_t` **`rtgui_view_event_handler`**(*struct rtgui_widget* widget, struct rtgui_event* event*)
  这个函数是viewr控件默认的事件处理函数。

`void` **`rtgui_view_set_box`**(*rtgui_view_t* view, rtgui_box_t* box*)
  这个函数用于为一个view控件设置自动布局用的box。

rtgui_modal_code_t **rtgui_view_show**(*rtgui_view_t* view, rt_bool_t is_modal*)
这个函数用于显示一个视图，参数is_modal用于指定视图是否显示成模态形式(即只有当获得用户确定的输入时，才从这个函数中退出)。如果是模态显示，返回值是最后用rtgui_view_end_modal设定的参数值；如果是非模态显示，返回值恒定为RTGUI_MODAL_OK。

void **rtgui_view_hide**(*rtgui_view_t* view*)
这个函数用于隐藏一个视图(仅针对非模态视图有效)。

void **rtgui_view_end_modal**(*rtgui_view_t* view, rtgui_modal_code_t modal_code*)
这个函数用于结束一个模态形式的视图，参数modal_code指定了退出模态显示的返回值。

void **rtgui_view_set_title**(*rtgui_view_t* view, const char* title*)
这个函数用于设置视图的标题。

char* rtgui_view_get_title(rtgui_view_t* view);()
这个函数用于获得视图的标题。

视图的例子：

```
/*
 * 程序清单：view演示
 *
 * 这是一个视图的演示，也是为了配合整个GUI演示而制作的视图，或者说，其他大多数控件的演示
 * 都是采用，先创建一个demo_view（演示视图），然后再在这个演示视图上添加相应的控件。
 *
 * 这个演示视图默认上方带一个演示标题，下方带两个按钮，点击它切换到前一个视图或后一个视图。
 * 针对控件演示而言，这个演示视图最重要的是提供了一个可显示的区域，只需要在这块区域上添加
 * 控件即可达到演示的目的。
 *
 * 获得这个显示区域的函数是：
 * demo_view_get_rect函数。
 */
#ifndef __DEMO_VIEW_H__
#define __DEMO_VIEW_H__

#include <rtgui/rtgui.h>
#include <rtgui/widgets/view.h>
#include <rtgui/widgets/workbench.h>

/* 如果是标准版本，可以启用box自动布局引擎 */
#ifndef RTGUI_USING_SMALL_SIZE
#include <rtgui/widgets/box.h>
#endif

/* 创建一个演示视图，需要给出这个视图所在的workbench和演示标题 */
rtgui_view_t* demo_view(rtgui_workbench_t* workbench, const char* title);
/* 获得演示视图提供给演示控件用的区域信息 */
void demo_view_get_rect(rtgui_view_t* view, rtgui_rect_t *rect);
void demo_view_show(void);

/* 如果是标准版，可以调用这个函数获得一个自动布局引擎 */
#ifndef RTGUI_USING_SMALL_SIZE
rtgui_box_t* demo_view_create_box(rtgui_view_t* view, int orient);
#endif
```

```

#endif
```

```
#include <rtgui/rtgui.h>
#include <rtgui/widgets/view.h>
#include <rtgui/widgets/button.h>
#include <rtgui/widgets/workbench.h>
#include <rtgui/widgets/staticline.h>

/* 用于存放演示视图的数组，最多可创建32个演示视图 */
static rtgui_view_t* demo_view_list[32];
/* 当前演示视图索引 */
static rt_uint16_t demo_view_current = 0;
/* 总共包括的演示视图数目 */
static rt_uint16_t demo_view_number = 0;

/* 显示下一个演示视图 */
void demo_view_next(struct rtgui_widget* widget, rtgui_event_t *event)
{
        if (demo_view_current + 1< demo_view_number)
        {
                demo_view_current ++;
                rtgui_view_show(demo_view_list[demo_view_current], RT_FALSE);
        }
}

/* 显示前一个演示视图 */
void demo_view_prev(struct rtgui_widget* widget, rtgui_event_t *event)
{
        if (demo_view_current != 0)
        {
                demo_view_current --;
                rtgui_view_show(demo_view_list[demo_view_current], RT_FALSE);
        }
}

/* 创建一个演示视图，需提供父workbench和演示用的标题 */
rtgui_view_t* demo_view(rtgui_workbench_t* workbench, const char* title)
{
        char view_name[32];
        struct rtgui_view* view;

        /* 设置视图的名称 */
        rt_sprintf(view_name, "view %d", demo_view_number + 1);
        view = rtgui_view_create(view_name);
        if (view == RT_NULL) return RT_NULL;

        /* 创建成功后，添加到数组中 */
        demo_view_list[demo_view_number] = view;
        demo_view_number ++;

        /* 添加到父workbench中 */
        rtgui_workbench_add_view(workbench, view);
```

```

        /* 添加下一个视图和前一个视图按钮 */
        {
                struct rtgui_rect rect;
                struct rtgui_button *next_btn, *prev_btn;
                struct rtgui_label *label;
                struct rtgui_staticline *line;

                /* 获得视图的位置信息(在加入到workbench中时, workbench会自动调整视图的大小) */
                rtgui_widget_get_rect(RTGUI_WIDGET(view), &rect);
                rect.x1 += 5;
                rect.y1 += 5;
                rect.x2 -= 5;
                rect.y2 = rect.y1 + 20;

                /* 创建标题用的标签 */
                label = rtgui_label_create(title);
                /* 设置标签位置信息 */
                rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
                /* 添加标签到视图中 */
                rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));

                rect.y1 += 20;
                rect.y2 += 20;
                /* 创建一个水平的staticline线 */
                line = rtgui_staticline_create(RTGUI_HORIZONTAL);
                /* 设置静态线的位置信息 */
                rtgui_widget_set_rect(RTGUI_WIDGET(line), &rect);
                /* 添加静态线到视图中 */
                rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(line));

                /* 获得视图的位置信息 */
                rtgui_widget_get_rect(RTGUI_WIDGET(view), &rect);
                rect.x2 -= 5;
                rect.y2 -= 5;
                rect.x1 = rect.x2 - 50;
                rect.y1 = rect.y2 - 20;

                /* 创建"下一个"按钮 */
                next_btn = rtgui_button_create("Next");
                /* 设置onbutton动作到demo_view_next函数 */
                rtgui_button_set_onbutton(next_btn, demo_view_next);
                /* 设置按钮的位置信息 */
                rtgui_widget_set_rect(RTGUI_WIDGET(next_btn), &rect);
                /* 添加按钮到视图中 */
                rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(next_btn));

                /* 获得视图的位置信息 */
                rtgui_widget_get_rect(RTGUI_WIDGET(view), &rect);
                rect.x1 += 5;
                rect.y2 -= 5;
                rect.x2 = rect.x1 + 50;
                rect.y1 = rect.y2 - 20;
```

```
104
105                 /* 创建"上一个"按钮 */
106                 prev_btn = rtgui_button_create("Prev");
107                 /* 设置onbutton动作到demo_view_prev函数 */
108                 rtgui_button_set_onbutton(prev_btn, demo_view_prev);
109                 /* 设置按钮的位置信息 */
110                 rtgui_widget_set_rect(RTGUI_WIDGET(prev_btn), &rect);
111                 /* 添加按钮到视图中 */
112                 rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(prev_btn));
113         }
114
115         /* 返回创建的视图 */
116         return view;
117 }
118
119 /* 这个函数用于返回演示视图的对外可用区域 */
120 void demo_view_get_rect(rtgui_view_t* view, rtgui_rect_t *rect)
121 {
122         RT_ASSERT(view != RT_NULL);
123         RT_ASSERT(rect != RT_NULL);
124
125         rtgui_widget_get_rect(RTGUI_WIDGET(view), rect);
126         /* 去除演示标题和下方按钮的区域 */
127         rect->y1 += 45;
128         rect->y2 -= 25;
129 }
130
131 /* 当是标准版本时，这个函数用于返回自动布局引擎box控件 */
132 #ifndef RTGUI_USING_SMALL_SIZE
133 rtgui_box_t* demo_view_create_box(rtgui_view_t* view, int orient)
134 {
135         rtgui_rect_t rect;
136         rtgui_box_t* box;
137
138         /* 获得视图的位置信息 */
139         rtgui_widget_get_rect(RTGUI_WIDGET(view), &rect);
140         rect.y1 += 45;
141         rect.y2 -= 25;
142
143         /* 创建一个自动布局引擎 */
144         box = rtgui_box_create(orient, &rect);
145         /* 添加box控件到视图中 */
146         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(box));
147
148         return box;
149 }
150 #endif
151
152 /* 这个函数用于显示当前的视图 */
153 void demo_view_show()
154 {
155         if (demo_view_number != 0)
156         {
```

```
157                 rtgui_view_show(demo_view_list[demo_view_current], RT_FALSE);
158         }
159 }
```

15.10.10 窗口：window

rtgui_win_t* **rtgui_win_create**(*rtgui_toplevel_t* parent_toplevel, const char* title, rtgui_rect_t *rect, rt_uint8_t flag*)

在创建窗口时，需要设定上层是否有父控件(通过parent_toplevel参数)。当有父控件存在时，其事件的上下文执行环境将依赖于父控件所属的线程。当parent_toplevel = RT_NULL时，那么这将是一个独立的窗口，具备独立的线程执行上下文。参数title指定窗口的标题；参数rect指定窗口在图形硬件环境中的坐标位置信息；flag指示出窗口的风格，当前支持：

- RTGUI_WIN_STYLE_MODAL – 模式窗口
- RTGUI_WIN_STYLE_CLOSED – 标题具备关闭按钮
- RTGUI_WIN_STYLE_ACTIVATE – 激活属性(这个属性用户不应该设置)
- RTGUI_WIN_STYLE_NO_FOCUS – 不具备获得焦点的窗口属性

void **rtgui_win_destroy**(*rtgui_win_t* win*)

这个函数用于删除一个窗口。

rtgui_modal_code_t **rtgui_win_show**(*rtgui_win_t* win, rt_bool_t is_modal*)

用于显示一个窗口，参数is_modal用于指定窗口是否显示成模态形式(即只有当获得用户确定的输入时，才从这个函数中退出)。如果是模态显示，返回值是最后用rtgui_win_end_modal设定的参数值；如果是非模态显示，返回值恒定为RTGUI_MODAL_OK。

void **rtgui_win_hiden**(*rtgui_win_t* win*)

这个函数用于隐藏一个窗口(仅针对非模态窗口有效)。

void **rtgui_win_end_modal**(*rtgui_win_t* win, rtgui_modal_code_t modal_code*)

函数用于结束一个模态形式的窗口，参数modal_code指定了退出模态显示的返回值。

rt_bool_t **rtgui_win_is_activated**(*struct rtgui_win* win*)

这个函数用于返回窗口是否处于激活状态。

void **rtgui_win_move**(*struct rtgui_win* win, int x, int y*)

这个函数用于移动窗口到指定位置。

void **rtgui_win_set_rect**(*rtgui_win_t* win, rtgui_rect_t* rect*)

这个函数用于设置窗口的位置信息。

void **rtgui_win_set_box**(*rtgui_win_t* win, rtgui_box_t* box*)

这个函数用于设置窗口的自动布局box。

void **rtgui_win_set_onactivate**(*rtgui_win_t* win, rtgui_event_handler_ptr handler*)

这个函数用于设置窗口的"onactivate"虚函数，这个虚函数将在窗口被激活时调用。

void **rtgui_win_set_ondeactivate**(*rtgui_win_t* win, rtgui_event_handler_ptr handler*)

这个函数用于设置窗口的"ondeactivate"虚函数，这个虚函数将在窗口被去激活时调用。

void **rtgui_win_set_onclose**(*rtgui_win_t* win, rtgui_event_handler_ptr handler*)

这个函数用于设置窗口的"onclose"虚函数，这个虚函数将在窗口被关闭时调用。

rt_bool_t **rtgui_win_event_handler**(*rtgui_widget_t* win, struct rtgui_event* event*)
  这个函数是viewr控件默认的事件处理函数。

void **rtgui_win_event_loop**(*rtgui_win_t* wnd*)
  这个函数是窗口的事件处理循环。当独立窗口线程启动后，应该执行这个函数以一直处理相应的窗口事件。

void **rtgui_win_set_title**(*rtgui_win_t* win, const char *title*)
  这个函数用于设置窗口的标题。

char* **rtgui_win_get_title**(*rtgui_win_t* win*)
  这个函数用于获得窗口的标题。

以下是一个使用window的例子：

```
/*
 * 程序清单：窗口演示
 *
 * 这个例子会先创建出一个演示用的view，当点击上面的按钮时会不同的模式创建窗口
 */

#include <rtgui/rtgui.h>
#include <rtgui/rtgui_system.h>
#include <rtgui/widgets/window.h>
#include <rtgui/widgets/label.h>
#include <rtgui/widgets/button.h>
#include "demo_view.h"

static struct rtgui_timer *timer;
static struct rtgui_label* label;
static struct rtgui_win* msgbox = RT_NULL;
static rt_uint8_t label_text[80];
static rt_uint8_t cnt = 5;

/* 获取一个递增的窗口标题 */
static char* get_win_title()
{
        static rt_uint8_t win_no = 0;
        static char win_title[16];

        rt_sprintf(win_title, "窗口 %d", ++win_no);
        return win_title;
}

/* 窗口关闭时的事件处理 */
void window_demo_close(struct rtgui_widget* widget, rtgui_event_t *even)
{
        rtgui_win_t* win;

        /* 获得最顶层控件 */
        win = RTGUI_WIN(rtgui_widget_get_toplevel(widget));

        /* 销毁窗口 */
        rtgui_win_destroy(win);
```

```
40  }
41
42  /* 关闭对话框时的回调函数 */
43  void diag_close(struct rtgui_timer* timer, void* parameter)
44  {
45          cnt --;
46          sprintf(label_text, "closed then %d second!", cnt);
47
48          /* 设置标签文本并更新控件 */
49          rtgui_label_set_text(label, label_text);
50          rtgui_widget_update(RTGUI_WIDGET(label));
51
52          if (cnt == 0)
53          {
54                  /* 超时，关闭对话框 */
55                  rtgui_win_destroy(msgbox);
56
57                  /* 停止并删除定时器 */
58                  rtgui_timer_stop(timer);
59                  rtgui_timer_destory(timer);
60          }
61  }
62
63  static rt_uint16_t delta_x = 20;
64  static rt_uint16_t delta_y = 40;
65
66  /* 触发正常窗口显示 */
67  static void demo_win_onbutton(struct rtgui_widget* widget, rtgui_event_t* event)
68  {
69          rtgui_win_t *win;
70          rtgui_label_t *label;
71          rtgui_toplevel_t *parent;
72          rtgui_rect_t rect = {0, 0, 150, 80};
73
74          parent = RTGUI_TOPLEVEL(rtgui_widget_get_toplevel(widget));
75          rtgui_rect_moveto(&rect, delta_x, delta_y);
76          delta_x += 20;
77          delta_y += 20;
78
79          /* 创建一个窗口 */
80          win = rtgui_win_create(parent,
81                  get_win_title(), &rect, RTGUI_WIN_STYLE_DEFAULT);
82
83          rect.x1 += 20;
84          rect.x2 -= 5;
85          rect.y1 += 5;
86          rect.y2 = rect.y1 + 20;
87
88          /* 添加一个文本标签 */
89          label = rtgui_label_create("这是一个普通窗口");
90          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
91          rtgui_container_add_child(RTGUI_CONTAINER(win), RTGUI_WIDGET(label));
92
```

```
93          /* 非模态显示窗口 */
94          rtgui_win_show(win, RT_FALSE);
95  }
96
97  /* 触发自动窗口显示 */
98  static void demo_autowin_onbutton(struct rtgui_widget* widget, rtgui_event_t* event)
99  {
100         rtgui_toplevel_t *parent;
101         struct rtgui_rect rect ={50, 50, 200, 200};
102
103         parent = RTGUI_TOPLEVEL(rtgui_widget_get_toplevel(widget));
104         msgbox = rtgui_win_create(parent, "Information", &rect, RTGUI_WIN_STYLE_DEFAULT);
105         if (msgbox != RT_NULL)
106         {
107                 cnt = 5;
108                 sprintf(label_text, "closed then %d second!", cnt);
109                 label = rtgui_label_create(label_text);
110                 rect.x1 += 5;
111                 rect.x2 -= 5;
112                 rect.y1 += 5;
113                 rect.y2 = rect.y1 + 20;
114                 rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
115                 rtgui_container_add_child(RTGUI_CONTAINER(msgbox), RTGUI_WIDGET(label));
116
117                 rtgui_win_show(msgbox, RT_FALSE);
118         }
119
120         /* 创建一个定时器 */
121         timer = rtgui_timer_create(100, RT_TIMER_FLAG_PERIODIC, diag_close, RT_NULL);
122         rtgui_timer_start(timer);
123 }
124
125 /* 触发模态窗口显示 */
126 static void demo_modalwin_onbutton(struct rtgui_widget* widget, rtgui_event_t* event)
127 {
128         rtgui_win_t *win;
129         rtgui_label_t *label;
130         rtgui_toplevel_t *parent;
131         rtgui_rect_t rect = {0, 0, 150, 80};
132
133         parent = RTGUI_TOPLEVEL(rtgui_widget_get_toplevel(widget));
134         rtgui_rect_moveto(&rect, delta_x, delta_y);
135         delta_x += 20;
136         delta_y += 20;
137
138         /* 创建一个窗口 */
139         win = rtgui_win_create(parent,
140                 get_win_title(), &rect, RTGUI_WIN_STYLE_DEFAULT);
141
142         rect.x1 += 20;
143         rect.x2 -= 5;
144         rect.y1 += 5;
145         rect.y2 = rect.y1 + 20;
```

```
146
147         label = rtgui_label_create("这是一个模式窗口");
148         rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
149         rtgui_container_add_child(RTGUI_CONTAINER(win), RTGUI_WIDGET(label));
150
151         /* 模态显示窗口 */
152         rtgui_win_show(win, RT_TRUE);
153         /* 采用模态显示窗口，关闭时不会自行删除窗口，需要主动删除窗口 */
154         rtgui_win_destroy(win);
155 }
156
157 /* 触发无标题窗口显示 */
158 static void demo_ntitlewin_onbutton(struct rtgui_widget* widget, rtgui_event_t* event)
159 {
160         rtgui_win_t *win;
161         rtgui_label_t *label;
162         rtgui_button_t *button;
163         rtgui_toplevel_t *parent;
164         rtgui_rect_t widget_rect, rect = {0, 0, 150, 80};
165
166         parent = RTGUI_TOPLEVEL(rtgui_widget_get_toplevel(widget));
167         rtgui_rect_moveto(&rect, delta_x, delta_y);
168         delta_x += 20;
169         delta_y += 20;
170
171         /* 创建一个窗口，风格为无标题及无边框 */
172         win = rtgui_win_create(parent,
173                 "no title", &rect, RTGUI_WIN_STYLE_NO_TITLE | RTGUI_WIN_STYLE_NO_BORDER);
174         RTGUI_WIDGET_BACKGROUND(RTGUI_WIDGET(win)) = white;
175
176         /* 创建一个文本标签 */
177         label = rtgui_label_create("无边框窗口");
178         rtgui_font_get_metrics(RTGUI_WIDGET_FONT(RTGUI_WIDGET(label)), "无边框窗口", &widget_rect);
179         rtgui_rect_moveto_align(&rect, &widget_rect, RTGUI_ALIGN_CENTER_HORIZONTAL);
180         widget_rect.y1 += 20;
181         widget_rect.y2 += 20;
182         rtgui_widget_set_rect(RTGUI_WIDGET(label), &widget_rect);
183         rtgui_container_add_child(RTGUI_CONTAINER(win), RTGUI_WIDGET(label));
184         RTGUI_WIDGET_BACKGROUND(RTGUI_WIDGET(label)) = white;
185
186         /* 创建一个关闭按钮 */
187         widget_rect.x1 = 0;
188         widget_rect.y1 = 0;
189         widget_rect.x2 = 40;
190         widget_rect.y2 = 20;
191         rtgui_rect_moveto_align(&rect, &widget_rect, RTGUI_ALIGN_CENTER_HORIZONTAL);
192         widget_rect.y1 += 40;
193         widget_rect.y2 += 40;
194         button = rtgui_button_create("关闭");
195         rtgui_widget_set_rect(RTGUI_WIDGET(button), &widget_rect);
196         rtgui_container_add_child(RTGUI_CONTAINER(win), RTGUI_WIDGET(button));
197         rtgui_button_set_onbutton(button, window_demo_close);
198
```

```
199         /* 非模态显示窗口 */
200         rtgui_win_show(win, RT_FALSE);
201 }
202 
203 rtgui_view_t* demo_view_window(rtgui_workbench_t* workbench)
204 {
205         rtgui_rect_t  rect;
206         rtgui_view_t* view;
207         rtgui_button_t *button;
208 
209         /* 创建一个演示用的视图 */
210         view = demo_view(workbench, "Window Demo");
211 
212         demo_view_get_rect(view, &rect);
213         rect.x1 += 5;
214         rect.x2 = rect.x1 + 100;
215         rect.y1 += 5;
216         rect.y2 = rect.y1 + 20;
217         /* 创建按钮用于显示正常窗口 */
218         button = rtgui_button_create("Normal Win");
219         rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
220         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
221         /* 设置onbutton为demo_win_onbutton函数 */
222         rtgui_button_set_onbutton(button, demo_win_onbutton);
223 
224         demo_view_get_rect(view, &rect);
225         rect.x1 += 5;
226         rect.x2 = rect.x1 + 100;
227         rect.y1 += 5 + 25;
228         rect.y2 = rect.y1 + 20;
229         /* 创建按钮用于显示一个自动关闭的窗口 */
230         button = rtgui_button_create("Auto Win");
231         rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
232         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
233         /* 设置onbutton为demo_autowin_onbutton函数 */
234         rtgui_button_set_onbutton(button, demo_autowin_onbutton);
235 
236         demo_view_get_rect(view, &rect);
237         rect.x1 += 5;
238         rect.x2 = rect.x1 + 100;
239         rect.y1 += 5 + 25 + 25;
240         rect.y2 = rect.y1 + 20;
241         /* 创建按钮用于触发一个模式窗口 */
242         button = rtgui_button_create("Modal Win");
243         rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
244         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
245         /* 设置onbutton为demo_modalwin_onbutton函数 */
246         rtgui_button_set_onbutton(button, demo_modalwin_onbutton);
247 
248         demo_view_get_rect(view, &rect);
249         rect.x1 += 5;
250         rect.x2 = rect.x1 + 100;
251         rect.y1 += 5 + 25 + 25 + 25;
```

```
252         rect.y2 = rect.y1 + 20;
253         /* 创建按钮用于触发一个不包含标题的窗口 */
254         button = rtgui_button_create("NoTitle Win");
255         rtgui_widget_set_rect(RTGUI_WIDGET(button), &rect);
256         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(button));
257         /* 设置onbutton为demo_ntitlewin_onbutton函数 */
258         rtgui_button_set_onbutton(button, demo_ntitlewin_onbutton);
259 
260         return view;
261 }
```

15.10.11 列表视图

列表视图提供的是一个个列表，这些列表可以分别对应到不同的功能上，类似于PC上的菜单的功能。当在相应的列表项上输入enter键时，将自动调用绑定在这个列表项上的回调函数。

列表项的定义如下：

```
struct rtgui_list_item
{
    char* name;
    rtgui_image_t *image;

    item_action action;
    void *parameter;
};
```

结构的解释如下

域	说明
name	列表项名称，即列表项显示时的标签
image	列表项图标
action	列表项绑定的动作
parameter	列表项绑定动作时输入的参数

Note: action函数的声明形式如下：

```
typedef void (*item_action)(void* parameter);
```

`rtgui_list_view_t*` **`rtgui_list_view_create`**(*const struct rtgui_list_item* items, rt_uint16_t count, rtgui_rect_t *rect*)

`void` **`rtgui_list_view_destroy`**(*rtgui_list_view_t* view*)

这两个函数分别用于创建或删除一个列表视图对象。创建时，参数items指示出列表项，count指示出存在多少个列表项，rect指示出view的位置信息。

`rt_bool_t` **`rtgui_list_view_event_handler`**(*struct rtgui_widget* widget, struct rtgui_event* event*)

这个函数是列表视图的默认事件处理函数

列表视图的使用例程：

```
/*
 * 程序清单：列表视图演示
 *
 * 这个例子会先创建出一个演示用的view，当点击上面的按钮时会按照模式显示的形式显示
 * 新的列表视图
 */
#include "demo_view.h"
#include <rtgui/widgets/label.h>
#include <rtgui/widgets/button.h>
#include <rtgui/widgets/window.h>
#include <rtgui/widgets/list_view.h>

static rtgui_workbench_t* workbench = RT_NULL;
static rtgui_list_view_t* _view = RT_NULL;
static rtgui_image_t* return_image = RT_NULL;

/* 列表项的动作函数 */
static void listitem_action(void* parameter)
{
        char label_text[32];
        rtgui_win_t *win;
        rtgui_label_t *label;
        rtgui_rect_t rect = {0, 0, 150, 80};
        int no = (int)parameter;

        rtgui_rect_moveto(&rect, 20, 50);

        /* 显示消息窗口 */
        win = rtgui_win_create(RTGUI_TOPLEVEL(workbench),
                "窗口", &rect, RTGUI_WIN_STYLE_DEFAULT);

        rect.x1 += 20;
        rect.x2 -= 5;
        rect.y1 += 5;
        rect.y2 = rect.y1 + 20;

        /* 添加相应的标签 */
        rt_sprintf(label_text, "动作 %d", no);
        label = rtgui_label_create(label_text);

        rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
        rtgui_container_add_child(RTGUI_CONTAINER(win), RTGUI_WIDGET(label));

        /* 非模态显示窗口 */
        rtgui_win_show(win, RT_FALSE);
}

/* 返回功能的动作函数 */
static void return_action(void* parameter)
{
        if (_view != RT_NULL)
        {
                /* 删除列表视图 */
```

```
54                  rtgui_view_destroy(RTGUI_VIEW(_view));
55                  _view = RT_NULL;
56          }
57  }
58
59  /* 各个列表项定义 */
60  static struct rtgui_list_item items[] =
61  {
62          {"列表项1", RT_NULL, listitem_action, (void*)1},
63          {"列表项2", RT_NULL, listitem_action, (void*)2},
64          {"列表项3", RT_NULL, listitem_action, (void*)3},
65          {"列表项4", RT_NULL, listitem_action, (void*)4},
66          {"列表项5", RT_NULL, listitem_action, (void*)5},
67          {"返回",    RT_NULL, return_action,    RT_NULL},
68  };
69
70  /* 打开列表视图用的按钮触发函数 */
71  static void open_btn_onbutton(rtgui_widget_t* widget, struct rtgui_event* event)
72  {
73          rtgui_rect_t rect;
74
75          /* 获得顶层的workbench */
76          workbench = RTGUI_WORKBENCH(rtgui_widget_get_toplevel(widget));
77          rtgui_widget_get_rect(RTGUI_WIDGET(workbench), &rect);
78
79          /* 创建一个列表视图, 项指定为items */
80          _view = rtgui_list_view_create(items, sizeof(items)/sizeof(struct rtgui_list_item),
81                  &rect);
82          /* 在workbench中添加相应的视图 */
83          rtgui_workbench_add_view(workbench, RTGUI_VIEW(_view));
84
85          /* 模式显示视图 */
86          rtgui_view_show(RTGUI_VIEW(_view), RT_FALSE);
87  }
88
89  /* 创建用于演示列表视图的视图 */
90  rtgui_view_t* demo_listview_view(rtgui_workbench_t* workbench)
91  {
92          rtgui_rect_t rect;
93          rtgui_view_t *view;
94          rtgui_button_t* open_btn;
95
96          view = demo_view(workbench, "列表视图演示");
97
98          /* 添加动作按钮 */
99          demo_view_get_rect(view, &rect);
100         rect.x1 += 5;
101         rect.x2 = rect.x1 + 80;
102         rect.y1 += 30;
103         rect.y2 = rect.y1 + 20;
104         open_btn = rtgui_button_create("打开列表");
105         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(open_btn));
106         rtgui_widget_set_rect(RTGUI_WIDGET(open_btn), &rect);
```

```
107         rtgui_button_set_onbutton(open_btn, open_btn_onbutton);
108 
109         return view;
110 }
```

15.10.12 文件列表视图

文件列表视图用于浏览目录、文件。当在文件上按enter键时，文件列表视图返回相应的文件名。

rtgui_filelist_view_t* **rtgui_filelist_view_create**(*rtgui_workbench_t* workbench, const char* directory, const char* pattern, const rtgui_rect_t* rect*)

void **rtgui_filelist_view_destroy**(*rtgui_filelist_view_t* view*)

这两个函数分别用于创建或删除一个文件列表视图对象。在创建时，workbench指示出了它的父workbench，directory指示出显示的目录，pattern指示出文件匹配的类型。

rt_bool_t **rtgui_filelist_view_event_handler**(*struct rtgui_widget* widget, struct rtgui_event* event*)

这个函数是文件列表视图的默认事件处理函数。

void **rtgui_filelist_view_set_directory**(*rtgui_filelist_view_t* view, const char* directory*)

这个函数用于设置文件列表视图的当前路径。

void **rtgui_filelist_get_fullpath**(*rtgui_filelist_view_t* view, char* path, rt_size_t len*)

这个函数用于获得当前文件的完整绝对路径，即路径名 + 文件名。

文件列表视图的例程如下：

```
1  /*
2   * 程序清单：文件列表视图演示
3   *
4   * 这个例子会先创建出一个演示用的view，当点击上面的按钮时会按照模式显示的形式显示
5   * 新的文件列表视图。
6   */
7  #include "demo_view.h"
8  #include <rtgui/widgets/label.h>
9  #include <rtgui/widgets/button.h>
10 #include <rtgui/widgets/filelist_view.h>
11 
12 /* 用于显示选择文件名的文本标签 */
13 static rtgui_label_t* label;
14 /* 触发文件列表视图的按钮回调函数 */
15 static void open_btn_onbutton(rtgui_widget_t* widget, struct rtgui_event* event)
16 {
17         rtgui_filelist_view_t *view;
18         rtgui_workbench_t *workbench;
19         rtgui_rect_t rect;
20 
21         /* 获得顶层的workbench对象 */
22         workbench = RTGUI_WORKBENCH(rtgui_widget_get_toplevel(widget));
23         rtgui_widget_get_rect(RTGUI_WIDGET(workbench), &rect);
24 
25         /* 针对Win32平台和其他平台做的不同的其实目录位置 */
```

```
26  #ifdef _WIN32
27          view = rtgui_filelist_view_create(workbench, "d:\\", "*.*", &rect);
28  #else
29          view = rtgui_filelist_view_create(workbench, "/", "*.*", &rect);
30  #endif
31
32          /* 采用模式形式显示文件列表视图 */
33          if (rtgui_view_show(RTGUI_VIEW(view), RT_TRUE) == RTGUI_MODAL_OK)
34          {
35                  char path[32];
36
37                  /* 在文件列表视图中成功选择文件，这里获得相应的路径名 */
38                  rtgui_filelist_get_fullpath(view, path, sizeof(path));
39
40                  /* 设置文件路径到文本标签 */
41                  rtgui_label_set_text(label, path);
42          }
43
44          /* 删除 文件列表 视图 */
45          rtgui_view_destroy(RTGUI_VIEW(view));
46  }
47
48  /* 创建用于演示文件列表视图的视图 */
49  rtgui_view_t* demo_fn_view(rtgui_workbench_t* workbench)
50  {
51          rtgui_rect_t rect;
52          rtgui_view_t* view;
53          rtgui_button_t* open_btn;
54          rtgui_font_t* font;
55
56          /* 默认采用12字体的显示 */
57          font = rtgui_font_refer("asc", 12);
58
59          /* 创建演示用的视图 */
60          view = demo_view(workbench, "FileList View");
61          /* 获得演示视图的位置信息 */
62          demo_view_get_rect(view, &rect);
63
64          rect.x1 += 5;
65          rect.x2 -= 5;
66          rect.y1 += 5;
67          rect.y2 = rect.y1 + 20;
68          /* 创建显示文件路径用的文本标签 */
69          label = rtgui_label_create("fn: ");
70          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(label));
71          rtgui_widget_set_rect(RTGUI_WIDGET(label), &rect);
72          RTGUI_WIDGET_FONT(RTGUI_WIDGET(label)) = font;
73
74          /* 获得演示视图的位置信息 */
75          demo_view_get_rect(view, &rect);
76          rect.x1 += 5;
77          rect.x2 = rect.x1 + 80;
78          rect.y1 += 30;
```

```
79          rect.y2 = rect.y1 + 20;
80          /* 创建按钮以触发一个新的文件列表视图 */
81          open_btn = rtgui_button_create("Open File");
82          rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(open_btn));
83          rtgui_widget_set_rect(RTGUI_WIDGET(open_btn), &rect);
84          RTGUI_WIDGET_FONT(RTGUI_WIDGET(open_btn)) = font;
85          rtgui_button_set_onbutton(open_btn, open_btn_onbutton);
86
87          return view;
88  }
```

15.10.13 实现自定义控件

以下是一个实现自定义mywidget的例子：

```
1   /*
2    * 程序清单：自定义控件
3    *
4    * 这个例子是要实现一个自定义控件，外观大致如
5    *   |
6    * --o--
7    *   |
8    * 的形状，中间的o色彩表示了当前的状态，ON状态时是绿色，OFF状态时是红色。
9    * 并且，这个o位置接受鼠标点击，点击下切换下相应的状态。
10   */
11  #ifndef __MY_WIDGET_H__
12  #define __MY_WIDGET_H__
13
14  #include <rtgui/rtgui.h>
15  #include <rtgui/widgets/widget.h>
16
17  /* 自定义控件的状态值定义 */
18  #define MYWIDGET_STATUS_ON        1
19  #define MYWIDGET_STATUS_OFF       0
20
21  /** 每个控件会有一个类型，通过如下的宏获得控件相应的类型信息 */
22  #define RTGUI_MYWIDGET_TYPE       (rtgui_mywidget_type_get())
23  /** 对一个对象实例，可以通过下面的宏实现类型转换 */
24  #define RTGUI_MYWIDGET(obj)       (RTGUI_OBJECT_CAST((obj), RTGUI_MYWIDGET_TYPE, rtgui_mywidget_t))
25  /** 可以通过下面的宏以决定一个具体实例是否是自定义控件类型 */
26  #define RTGUI_IS_MYWIDGET(obj)    (RTGUI_OBJECT_CHECK_TYPE((obj), RTGUI_MYWIDGET_TYPE))
27
28  /* 个性化控件类定义 */
29  struct rtgui_mywidget
30  {
31          /* 这个控件是继承自rtgui_widget控件 */
32          struct rtgui_widget parent;
33
34          /* 状态：ON、OFF */
35          rt_uint8_t status;
36  };
37  typedef struct rtgui_mywidget rtgui_mywidget_t;
```

```
38
39  /* 这个函数用于获得自定义控件的类型 */
40  rtgui_type_t *rtgui_mywidget_type_get(void);
41
42  /* 控件的创建和删除 */
43  struct rtgui_mywidget* rtgui_mywidget_create(rtgui_rect_t* r);
44  void rtgui_mywidget_destroy(struct rtgui_mywidget* me);
45
46  /* 控件的默认事件处理函数。
47   * 对一个控件而言，如果派生自它的子控件很可能会调用父控件的事件处理函数，
48   * 所以这里采用公开声明的方式。
49   */
50  rt_bool_t rtgui_mywidget_event_handler(struct rtgui_widget* widget, struct rtgui_event* event);
51
52  #endif
```

```
1   #include <rtgui/dc.h>
2   #include "mywidget.h"
3
4   /* 控件绘图函数 */
5   static void rtgui_mywidget_ondraw(struct rtgui_mywidget* me)
6   {
7           struct rtgui_dc* dc;
8           struct rtgui_rect rect;
9           rt_uint16_t x, y;
10
11          /* 获得目标DC，开始绘图 */
12          dc = rtgui_dc_begin_drawing(RTGUI_WIDGET(me));
13          if (dc == RT_NULL) return;
14
15          /* 获得窗口的尺寸 */
16          rtgui_widget_get_rect(RTGUI_WIDGET(me), &rect);
17          /* 绘制背景色 */
18          rtgui_dc_set_color(dc, white);
19          rtgui_dc_fill_rect(dc, &rect);
20
21          /* 计算中心原点 */
22          x = (rect.x2 + rect.x1)/2;
23          y = (rect.y2 + rect.y1)/2;
24
25          /* 绘制十字架 */
26          rtgui_dc_set_color(dc, black);
27          rtgui_dc_draw_hline(dc, rect.x1, rect.x2, y);
28          rtgui_dc_draw_vline(dc, x, rect.y1, rect.y2);
29
30          /* 根据状态绘制圆圈 */
31          if (me->status == MYWIDGET_STATUS_ON)
32                  rtgui_dc_set_color(dc, green);
33          else
34                  rtgui_dc_set_color(dc, red);
35          rtgui_dc_fill_circle(dc, x, y, 5);
36
37          /* 结束绘图 */
```

```
38          rtgui_dc_end_drawing(dc);
39          return;
40  }
41
42  /* 鼠标事件处理函数 */
43  static void rtgui_mywidget_onmouse(struct rtgui_mywidget* me, struct rtgui_event_mouse* mouse)
44  {
45          struct rtgui_rect rect;
46          rt_uint16_t x, y;
47
48          /* 仅对鼠标抬起动作进行处理 */
49          if (!(mouse->button & RTGUI_MOUSE_BUTTON_UP)) return;
50
51          /* 获得控件的位置 */
52          rtgui_widget_get_rect(RTGUI_WIDGET(me), &rect);
53          /* get_rect函数获得是控件的相对位置，而鼠标事件给出的坐标是绝对坐标，需要做一个转换 */
54          rtgui_widget_rect_to_device(RTGUI_WIDGET(me), &rect);
55
56          /* 计算中心原点 */
57          x = (rect.x2 + rect.x1)/2;
58          y = (rect.y2 + rect.y1)/2;
59
60          /* 比较鼠标坐标是否在圈内 */
61          if ((mouse->x < x + 5 && mouse->x > x - 5) &&
62                          (mouse->y < y + 5 && mouse->y > y - 5))
63          {
64                  /* 更改控件状态 */
65                  if (me->status & MYWIDGET_STATUS_ON) me->status = MYWIDGET_STATUS_OFF;
66                  else me->status = MYWIDGET_STATUS_ON;
67
68                  /* 刷新(重新绘制)控件 */
69                  rtgui_mywidget_ondraw(me);
70          }
71  }
72
73  /* mywidget控件的事件处理函数 */
74  rt_bool_t rtgui_mywidget_event_handler(struct rtgui_widget* widget, struct rtgui_event* event)
75  {
76          /* 调用事件处理函数时，widget指针指向控件本身，所以先获得相应控件对象的指针 */
77          struct rtgui_mywidget* me = RTGUI_MYWIDGET(widget);
78
79          switch (event->type)
80          {
81          case RTGUI_EVENT_PAINT:
82                  /* 绘制事件，调用绘图函数绘制 */
83                  rtgui_mywidget_ondraw(me);
84                  break;
85
86          case RTGUI_EVENT_MOUSE_BUTTON:
87                  /* 鼠标事件 */
88                  rtgui_mywidget_onmouse(RTGUI_MYWIDGET(me), (struct rtgui_event_mouse*) event);
89                  break;
90
```

```
91                  /* 其他事件调用父类的事件处理函数 */
92          default:
93                  return rtgui_widget_event_handler(widget, event);
94          }
95  
96          return RT_FALSE;
97  }
98  
99  /* 自定义控件的构造函数 */
100 static void _rtgui_mywidget_constructor(rtgui_mywidget_t *mywidget)
101 {
102         /* 默认这个控件接收聚焦 */
103         RTGUI_WIDGET(mywidget)->flag |= RTGUI_WIDGET_FLAG_FOCUSABLE;
104         /* 初始化控件并设置事件处理函数 */
105         rtgui_widget_set_event_handler(RTGUI_WIDGET(mywidget), rtgui_mywidget_event_handler);
106 
107         /* 初始状态时OFF */
108         mywidget->status = MYWIDGET_STATUS_OFF;
109 }
110 
111 /* 获得控件的类型 */
112 rtgui_type_t *rtgui_mywidget_type_get(void)
113 {
114         /* 控件的类型是一个静态变量，默认是NULL */
115         static rtgui_type_t *mywidget_type = RT_NULL;
116 
117         if (!mywidget_type)
118         {
119                 /* 当控件类型不存在时，创建它，并指定这种类型数据的大小及指定相应的构造函数和析构函数 */
120                 mywidget_type = rtgui_type_create("mywidget", RTGUI_WIDGET_TYPE,
121                         sizeof(rtgui_mywidget_t),
122                         RTGUI_CONSTRUCTOR(_rtgui_mywidget_constructor), RT_NULL);
123         }
124 
125         return mywidget_type;
126 }
127 
128 /* 创建一个自定义控件 */
129 struct rtgui_mywidget* rtgui_mywidget_create(rtgui_rect_t* r)
130 {
131         struct rtgui_mywidget* me;
132 
133         /* 让rtgui_widget创建出一个指定类型：RTGUI_MYWIDGET_TYPE类型的控件 */
134         me = (struct rtgui_mywidget*) rtgui_widget_create (RTGUI_MYWIDGET_TYPE);
135         if (me != RT_NULL)
136         {
137                 rtgui_widget_set_rect(RTGUI_WIDGET(me), r);
138         }
139 
140         return me;
141 }
142 
143 /* 删除一个自定义控件 */
```

```
144 void rtgui_mywidget_destroy(struct rtgui_mywidget* me)
145 {
146         rtgui_widget_destroy(RTGUI_WIDGET(me));
147 }
```

```
1  /*
2   * 程序清单：自定义控件演示
3   *
4   * 这个例子会在创建出的view上添加两个自定义控件
5   */
6  #include "demo_view.h"
7  #include "mywidget.h"
8
9  /* 创建用于演示自定义控件的视图 */
10 rtgui_view_t *demo_view_mywidget(rtgui_workbench_t* workbench)
11 {
12         rtgui_view_t *view;
13         rtgui_rect_t rect;
14         rtgui_mywidget_t *mywidget;
15
16         /* 先创建一个演示用的视图 */
17         view = demo_view(workbench, "MyWidget View");
18
19         /* 获得视图的位置信息 */
20         demo_view_get_rect(view, &rect);
21         rect.x1 += 5;
22         rect.x2 = rect.y1 + 80;
23         rect.y1 += 5;
24         rect.y2 = rect.y1 + 80;
25         /* 创建一个自定义控件 */
26         mywidget = rtgui_mywidget_create(&rect);
27         /* view是一个container控件，调用add_child方法添加这个自控件 */
28         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(mywidget));
29
30         /* 获得视图的位置信息 */
31         demo_view_get_rect(view, &rect);
32         rect.x1 += 25;
33         rect.x2 = rect.y1 + 40;
34         rect.y1 += 5 + 100;
35         rect.y2 = rect.y1 + 40;
36         /* 创建一个自定义控件 */
37         mywidget = rtgui_mywidget_create(&rect);
38         /* view是一个container控件，调用add_child方法添加这个自控件 */
39         rtgui_container_add_child(RTGUI_CONTAINER(view), RTGUI_WIDGET(mywidget));
40
41         return view;
42 }
```

CHAPTER

SIXTEEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

APPENDIX

A

内核配置

RT-Thread内核是一个可配置的内核，可根据选项的不同以支持不同的特性。RT-Thread的内核是由rtconfig.h头文件控制。

A.1 rtconfig.h配置头文件

可以通过修改rtconfig.h的方式来修改RT-Thread系统的配置。下面是RT-Thread/STM32F103ZE的配置头文件。

```
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* 系统中对象名称大小 */
#define RT_NAME_MAX     8

/* 对齐方式 */
#define RT_ALIGN_SIZE   4

/* 最大支持的优先级：32或256 */
#define RT_THREAD_PRIORITY_MAX  256

/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100

/* SECTION: 调试选项 */
/* 调试 */
#define RT_THREAD_DEBUG

/* 线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE
```

```
/* 支持互斥锁 */
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
#define RT_CONSOLEBUF_SIZE      128

/* SECTION: FinSH shell 选项 */
/* 支持finsh作为shell */
#define RT_USING_FINSH
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
/* 使用小型libc库 */
/* #define RT_USING_MINILIBC */

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

/* 支持RTGUI */
```

```
/* #define RT_USING_RTGUI */

/* SECTION: 设备虚拟文件系统 */
#define RT_USING_DFS
/* 支持最大的文件系统数目 */
#define DFS_FILESYSTEMS_MAX             2
/* 最大同时打开文件数               */
#define DFS_FD_MAX                      8
/* 最大扇区缓冲数目                 */
#define DFS_CACHE_MAX_NUM               8

/* SECTION: 轻型TCP/IP协议栈选项 */
/* 支持LwIP协议栈 */
#define RT_USING_LWIP
/* 支持WebServer */
#define RT_USING_WEBSERVER

/* 打开LwIP调试信息 */
/* #define RT_LWIP_DEBUG */

/* 使能ICMP协议 */
#define RT_LWIP_ICMP

/* 使能IGMP协议 */
/* #define RT_LWIP_IGMP */

/* 使能 UDP 协议/
#define RT_LWIP_UDP

/* 使能 TCP protocol*/
#define RT_LWIP_TCP

/* 同时支持的TCP连接数 */
#define RT_LWIP_TCP_PCB_NUM             5

/* TCP发送缓冲空间 */
#define RT_LWIP_TCP_SND_BUF             1500

/* 使能 SNMP 协议 */
/* #define RT_LWIP_SNMP */

/* 使能 DHCP*/
/* #define RT_LWIP_DHCP */

/* 使能 DNS */
#define RT_LWIP_DNS

/* 本机IP地址 */
#define RT_LWIP_IPADDR0 192
#define RT_LWIP_IPADDR1 168
#define RT_LWIP_IPADDR2 1
#define RT_LWIP_IPADDR3 30
```

```
/* 网关地址 */
#define RT_LWIP_GWADDR0 192
#define RT_LWIP_GWADDR1 168
#define RT_LWIP_GWADDR2 1
#define RT_LWIP_GWADDR3 1

/* 本机掩码地址 */
#define RT_LWIP_MSKADDR0                255
#define RT_LWIP_MSKADDR1                255
#define RT_LWIP_MSKADDR2                255
#define RT_LWIP_MSKADDR3                0

/* TCP线程选项 */
#define RT_LWIP_TCPTHREAD_PRIORITY      120
#define RT_LWIP_TCPTHREAD_MBOX_SIZE     4
#define RT_LWIP_TCPTHREAD_STACKSIZE     1024

/* 以太网线程选项 */
#define RT_LWIP_ETHTHREAD_PRIORITY      128
#define RT_LWIP_ETHTHREAD_MBOX_SIZE     4
#define RT_LWIP_ETHTHREAD_STACKSIZE     512

#endif
```

APPENDIX B

ARM基本知识

本章及后面的两章讲述的是RT-Thread的ARM移植，分别介绍AT91SAM7S64在GNU GCC环境下和RealView MDK环境下的移植。ATMEL AT91SAM7S64这款芯片采用了ARM7TDMI架构，在进行移植之前很有必要对ARM的编程模式进行详细的了解。

B.1 ARM的工作状态

从编程的角度看，ARM微处理器的工作状态一般有两种，并可在两种状态之间切换：

- 第一种为ARM状态，此时处理器执行32位的字对齐的ARM指令；
- 第二种为Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

当ARM微处理器执行32位的ARM指令集时，工作在ARM状态；当ARM微处理器执行16位的Thumb指令集时，工作在Thumb状态。在程序的执行过程中，微处理器可以随时在两种工作状态之间切换，并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。采用Thumb指令，生成的代码体积相对ARM指令要小，但Thumb指令也有一些局限性，例如它并不标准的程序调用栈，从软件上很难实现栈的回溯等。

在目前RT-Thread对ARM的支持上，RT-Thread只能工作于ARM状态。

B.2 ARM处理器模式

ARM微处理器支持7种运行模式，分别为：

ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

在用户模式下，支持的指令集是受限的，需要使用特权指令时需要通过一定的方式（例如软中断）切换到系统模式进行处理。这对RTOS来说，模式的切换增加了不必要的开销，所以绝大多数RTOS使用了系统模式。同样RT-Thread应用线程选择的是运行于系统模式，这样进行系统函数调用时可不用通过软中断方式陷入到系统模式中。

B.3 ARM的寄存器组织

因为目前RT-Thread还不支持运行Thumb状态，所以本节主要说明ARM状态下的寄存器组织。

ARM体系结构中包括通用寄存器和特殊寄存器。通用寄存器包括R0～R15，可以分为三类：

未分组寄存器R0～R7； 分组寄存器R8～R14 程序计数器PC(R15) 未分组寄存器R0～R7在所有运行模式中，代码中指向的寄存器在物理上都是 唯一 的，他们未被系统用作特殊的用途，因此在中断或异常处理进行运行模式切换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏。

分组寄存器R8～R14则是和运行模式相关，代码中指向的寄存器和处理器当前运行的模式密切相关。

对于R8～R12来说，每个寄存器对应 两个不同的物理寄存器 ，当使用FIQ模式时，访问寄存器R8_fiq～R12_fiq；当使用除FIQ模式以外的其他模式时，访问寄存器R8_usr～R12_usr。

对于R13、R14来说，每个寄存器对应 **6**个不同的物理寄存器 ，其中的一个是用户模式与系统模式共用，另外5个物理寄存器对应于其他5种不同的运行模式。采用以下的记号来区分不同的物理寄存器：

- R13_<mode>
- R14_<mode>

其中，mode为以下几种模式之一：usr、fiq、irq、svc、abt、und。

由于处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分都需要初始化每种模式下的R13，使其指向该运行模式的栈空间，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14也称作子程序连接寄存器（Subroutine Link Register）或连接寄存器LR。当执行**BL子程序调用指令**时，R14中得到R15（程序计数器PC）的备份。其他情况下，R14用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器R14_svc、R14_irq、R14_fiq、R14_abt和R14_und用来保存R15的返回值。

寄存器R14常用在如下的情况：

在每一种运行模式下，都可用R14保存子程序的返回地址，当用BL或BLX指令调用子程序时，将PC的当前值拷贝给R14。执行完子程序后，又将R14的值拷贝回PC，即可完成子程序的调用返回。

程序计数器PC(R15)用作程序计数器（PC）。在ARM状态下，位[1:0]为0，位[31:2]用于保存PC；在Thumb状态下，位[0]为0，位[31:1]用于保存PC；在ARM状态下，PC的0和1位是0，在Thumb状态下，PC的0位是0。

CPSR(Current Program Status Register，当前程序状态寄存器)，CPSR可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR（Saved Program Status Register，备份的程序状态寄存器），当异常发生时，SPSR用于保存CPSR的当前值，从异常退出时则可由SPSR来恢复CPSR。

ARM状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM状态下的程序状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= 分组寄存器

B.4 ARM的异常

当正常的程序执行流程发生暂时的停止时，称之为异常，例如处理一个外部的中断请求。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

当一个异常出现以后，ARM微处理器会执行以下几步操作：

- 将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从 ARM状态进入，LR寄存器中保存的是下一条指令的地址（当前PC＋4或PC＋8，与异常的类型有关）；若异常是从Thumb状态进入，则在LR寄存器中保存当前PC的偏移量，这样，异常处理程序就不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令 MOV PC，R14_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。
- 将CPSR复制到相应的SPSR中。
- 根据异常类型，强制设置CPSR的运行模式位。

- 强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

还可以设置中断禁止位，以禁止中断发生。如果异常发生时，处理器处于Thumb状态，则当异常向量地址加载入PC时，处理器自动切换到ARM状态。

ARM微处理器对异常的响应过程用伪码可以描述为：

```
R14_<Exception_Mode> = Return Link

SPSR_<Exception_Mode> = CPSR

CPSR[4:0] = Exception Mode Number

CPSR[5] = 0 ; 当运行于 ARM 工作状态时

If <Exception_Mode> == Reset or FIQ then ; 当响应 FIQ 异常时，禁止新的 FIQ 异常

CPSR[6] = 1

CPSR[7] = 1

PC = Exception Vector Address
```

异常处理完毕之后，ARM微处理器会执行以下几步操作从异常返回：

- 将连接寄存器LR的值减去相应的偏移量后送到PC中。
- 将SPSR复制回CPSR中。
- 若在进入异常处理时设置了中断禁止位，要在此清除。

可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。当系统运行时，异常可能会随时发生，为保证在 ARM 处理器发生异常时不至于处于未知状态，在应用程序的设计中，首先要进行异常处理，采用的方式是在异常向量表中的特定位置放置一条跳转指令，跳转到异常处理程序，当 ARM 处理器发生异常时，程序计数器 PC 会被强制设置为对应的异常向量，从而跳转到异常处理程序，当异常处理完成以后，返回到主程序继续执行。

B.5 ARM的IRQ中断处理

RT-Thread的ARM体系结构移植只涉及到IRQ中断，所以本节只讨论IRQ中断模式，主要包括ARM微处理器在硬件上是如何响应中断及如何从中断中返回的。

当中断产生时，ARM7TDMI将执行的操作

1. 把当前CPSR寄存器的内容拷贝到相应的SPSR寄存器。这样当前的工作模式、中断屏蔽位和状态标志被保存下来。
2. 转入相应的模式，并关闭IRQ中断。
3. 把PC值减4后，存入相应的LR寄存器。
4. 将PC寄存器指向IRQ中断向量位置。

由中断返回时，ARM7TDMI将完成的操作

1. 将备份程序状态寄存器的内容拷贝到当前程序状态寄存器，恢复中断前的状态。

2. 清除相应禁止中断位（如果已设置的话）。
3. 把连接寄存器的值拷贝到程序计数器，继续运行原程序。

B.6 AT91SAM7S64概述

AT91SAM7S64是ATMEL 32位ARM RISC处理器小引脚数Flash微处理器家族的一员。它包含一个ARM7TDMI高性能RISC核心，64KB片上高速flash（512页，每页包含128字节），16KB片内高速静态RAM，2个同步串口（USART），USB 2.0全速Device设备，3个16bit定时器/计数器通道等。

APPENDIX

C

GNU GCC移植

GNU GCC是GNU的多平台编译器，也是开源项目中的首选编译环境，支持ARM各个版本的指令集，MIPS，x86等多个体系结构，也为一些知名操作系统作为官方编译器（例如主流的几种BSD操作系统，Linux操作系统，vxWorks实时操作系统等），所以作为开源项目的RT-Thread首选编译器是GNU GCC，甚至在一些地方会对GCC做专门的优化。

以下就以AT91SAM7S核心板为例，描述如何进行RT-Thread的移植。

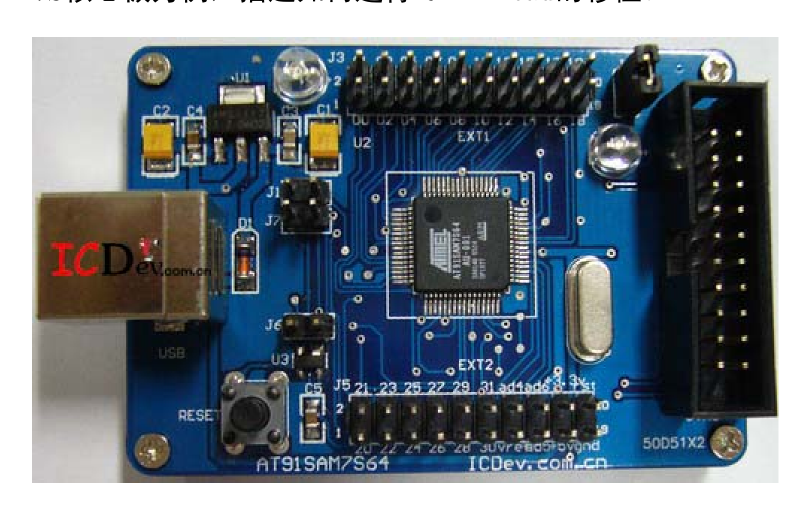


Figure C.1: AT91SAM7S64核心板(由icdev.net提供)

C.1 CPU相关移植

和通用平台中的GCC不同，编译操作系统会生成单独的目标文件，一些基本的算术操作例如除法，必须在链接的时候选择使用gcc库（libgcc.a），还是自身的实现。RT-Thread推荐选择后者：自己实现一些基本的算术操作，因为这样能够让生成的目标文件体积更小一些。这些基本的算术操作统一放在各自体系结构目录下的common目录。另外ARM体系结构中ARM模式下的一些过程调用也是标准的，所以也放置了一些栈回溯的代码例程（在Thumb模式下这部分代码将不可用）。

kernel/libcpu/arm/common目录下的文件

目前common目录下这些文件都已经存在，其他的ARM芯片移植基本上不需要重新实现或修改。

C.1.1 上下文切换代码

任务切换代码是移植一个操作系统首先要考虑的，因为它关系到线程间的正常运行，是核心中的核心。

在目录中添加context_gcc.S代码，代码如下

```
/*
 * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to)
 * 上下文切换函数，
 * r0寄存器指向保存当前线程栈位置
 * r1寄存器指向切换到线程的栈位置
 */
.globl rt_hw_context_switch
rt_hw_context_switch:
    stmfd   sp!, {lr}           /* 把LR寄存器压入栈，也就是从这个函数返回后的下一执行处 */
    stmfd   sp!, {r0-r12, lr}   /* 把R0 - R12以及LR压入栈 */

    mrs     r4, cpsr            /* 读取CPSR寄存器到R4寄存器 */
    stmfd   sp!, {r4}           /* 把R4寄存器压栈（即上一指令取出的CPSR寄存器）*/
    mrs     r4, spsr            /* 读取SPSR寄存器到R4寄存器 */
    stmfd   sp!, {r4}           /* 把R4寄存器压栈（即SPSR寄存器） */

    str     sp, [r0]            /* 把栈指针更新到TCB的sp，是由R0传入此函数 */

    /* 到这里换出线程的上下文都保存在栈中 */

    ldr     sp, [r1]            /* 载入切换到线程的TCB的sp，即此线程换出时保存的sp寄存器 */

    /* 从切换到线程的栈中恢复上下文，次序和保存的时候刚好相反 */
    ldmfd   sp!, {r4}           /* 出栈到R4寄存器（保存了SPSR寄存器） */
    msr     spsr_cxsf, r4      /* 恢复SPSR寄存器 */
    ldmfd   sp!, {r4}           /* 出栈到R4寄存器（保存了CPSR寄存器） */
    msr     cpsr_cxsf, r4      /* 恢复CPSR寄存器 */

    ldmfd   sp!, {r0-r12, lr, pc}   /* 对R0 - R12及LR、PC进行恢复 */

/*
 * void rt_hw_context_switch_to(rt_uint32 to)/*
 * 此函数只在系统进行第一次发生任务切换时使用，因为是从没有线程的状态进行切换
 * 实现上，刚好是rt_hw_context_switch的下半截
 */
.globl rt_hw_context_switch_to
rt_hw_context_switch_to:
    ldr     sp, [r0]            /* 获得切换到线程的SP指针 */

    ldmfd   sp!, {r4}           /* 出栈R4寄存器（保存了SPSR寄存器值） */
    msr     spsr_cxsf, r4      /* 恢复SPSR寄存器 */
    ldmfd   sp!, {r4}           /* 出栈R4寄存器（保存了CPSR寄存器值）*/
    msr     cpsr_cxsf, r4      /* 恢复CPSR寄存器 */

    ldmfd   sp!, {r0-r12, lr, pc}   /* 恢复R0 - R12，LR及PC寄存器 */
```

在RT-Thread中，如果中断服务例程触发了上下文切换（即执行了一次rt_schedule函数试图进行

一次调度），它会设置标志rt_thread_switch_interrupt_flag为真。而后在所有中断服务例程都处理完毕向线程返回的时候，如果rt_thread_switch_interrupt_flag为真，那么中断结束后就必须进行线程的上下文切换。这部分上下文切换代码和上面会有些不同，这部分在下一个文件中叙述，但设置rt_thread_switch_interrupt_flag标志的代码以及保存切换出和切换到线程的函数在这里给出，如下代码所示。

```
/*
 * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to)/*
 * 此函数会在调度器中调用，在调度器做上下文切换前会判断是否处于中断服务模式中，如果
 * 是则调用rt_hw_context_switch_interrupt函数（设置中断中任务切换标志）
 * 否则调用 rt_hw_context_switch函数（进行真正的线程上线文切换）
 */
    rt_hw_context_switch_interrupt:
            ldr r2, =rt_thread_switch_interrput_flag
            ldr r3, [r2]                            ; 载入中断中切换标致地址
            cmp r3, #1                              ; 等于 1 ?
            beq _reswitch                           ; 如果等于1，跳转到_reswitch
            mov r3, #1                              ; 设置中断中切换标志位1
            str r3, [r2]                            ; 保存到标志变量中
            ldr r2, =rt_interrupt_from_thread
            str r0, [r2]                            ; 保存切换出线程栈指针
    _reswitch:
            ldr r2, =rt_interrupt_to_thread
            str r1, [r2]                            ; 保存切换到线程栈指针
            bx      lr
```

上面的代码等价于C代码：

```
;/*
; * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
; */
rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to)
{
    if (rt_thread_switch_interrput_flag == 1)
        rt_interrupt_from_thread = from;
    else
        rt_thread_switch_interrput_flag = 1;

    rt_interrupt_to_thread = to;
}
```

除了上下文切换外，也在这个文件中实现了中断（IRQ & FIQ）的关闭和恢复（操作CPSR寄存器屏蔽/使能所有中断）。

```
/*
 * rt_base_t rt_hw_interrupt_disable()
 * 关闭IRQ和FIQ中断，返回关闭中断前的状态
 */
.globl rt_hw_interrupt_disable
rt_hw_interrupt_disable:
    mrs r0, cpsr        /* 保存CPSR寄存器的值到R0寄存器 */
    orr r1, r0, #0xc0   /* R0寄存器的值或上0xc0（2、3位置1），结果放到r1中 */
    msr cpsr_c, r1      /* 把R1的值存放到CPSR寄存器中 */
```

```
    mov pc, lr          /* 返回调用rt_hw_interrupt_disable函数处，返回值在R0中 */

/*
 * void rt_hw_interrupt_enable(rt_base_t level)
 * 恢复中断状态，中断状态在R0寄存器中
 */
.globl rt_hw_interrupt_enable
rt_hw_interrupt_enable:
    msr cpsr, r0        /* 把R0的值保存到CPSR中 */
    mov pc, lr          /* 函数返回 */
```

C.1.2 系统启动代码

接下来是系统启动的代码。因为ARM体系结构异常的触发总是置于0地址的（或者说异常向量表总是置于0地址），所以操作系统要捕获异常（最重要的是中断异常）就必须放置上自己的向量表。

此外，由于系统刚上电可能一些地方也需要进行初始化（RT-Thread推荐和板子相关的初始化最好放在bsp目录中），对ARM体系结构，另一个最重要的地方就是（各种模式下）栈的设置。下面的代码（文件start_gcc.S）列出了系统启动部分的汇编代码：

```
    /* AT91SAM7S64内部Memory基地址 */
    .equ    FLASH_BASE,     0x00100000
    .equ    RAM_BASE,       0x00200000

    /* 栈顶及各个栈大小的配置 */
    .equ    TOP_STACK,      0x00204000
    .equ    UND_STACK_SIZE, 0x00000000
    .equ    SVC_STACK_SIZE, 0x00000400
    .equ    ABT_STACK_SIZE, 0x00000000
    .equ    FIQ_STACK_SIZE, 0x00000100
    .equ    IRQ_STACK_SIZE, 0x00000100
    .equ    USR_STACK_SIZE, 0x00000004

    /* ARM模式的定义 */
    .equ    MODE_USR, 0x10
    .equ    MODE_FIQ, 0x11
    .equ    MODE_IRQ, 0x12
    .equ    MODE_SVC, 0x13
    .equ    MODE_ABT, 0x17
    .equ    MODE_UND, 0x1B
    .equ    MODE_SYS, 0x1F

    .equ    I_BIT, 0x80    /* IRQ位 */
    .equ    F_BIT, 0x40    /* FIQ位 */

.section .init, "ax"
.code 32
.align 0
.globl _start
_start:
    b   reset
```

```
    ldr pc, _vector_undef
    ldr pc, _vector_swi
    ldr pc, _vector_pabt
    ldr pc, _vector_dabt
    nop                         /* 保留的异常项 */
    ldr pc, _vector_irq
    ldr pc, _vector_fiq

_vector_undef:  .word vector_undef
_vector_swi:    .word vector_swi
_vector_pabt:   .word vector_pabt
_vector_dabt:   .word vector_dabt
_vector_resv:   .word vector_resv
_vector_irq:    .word vector_irq
_vector_fiq:    .word vector_fiq

/*
 * RT-Thread BSS段起始、结束位置，这个在链接脚本中定义
 */
.globl _bss_start
_bss_start: .word __bss_start
.globl _bss_end
_bss_end:   .word __bss_end

/* 系统入口 */
reset:
    /* 关闭看门狗 */
    ldr r0, =0xFFFFFD40
    ldr r1, =0x00008000
    str r1, [r0, #0x04]

    /* 使能主晶振 */
    ldr r0, =0xFFFFFC00
    ldr r1, =0x00000601
    str r1, [r0, #0x20]

    /* 等待晶振稳定 */
moscs_loop:
    ldr r2, [r0, #0x68]
    ands r2, r2, #1
    beq moscs_loop

    /* 设置PLL */
    ldr r1, =0x00191C05
    str r1, [r0, #0x2C]

    /* 等待PLL上锁 */
pll_loop:
    ldr r2, [r0, #0x68]
    ands r2, r2, #0x04
    beq pll_loop

    /* 选择clock */
```

```
    ldr r1, =0x00000007
    str r1, [r0, #0x30]

    /* 设置各个模式下的栈 */
    ldr r0, =TOP_STACK

    /* 设置栈 */
    /* undefined模式 */
    msr cpsr_c, #MODE_UND|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #UND_STACK_SIZE

    /* abort模式 */
    msr cpsr_c, #MODE_ABT|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #ABT_STACK_SIZE

    /* FIQ模式 */
    msr cpsr_c, #MODE_FIQ|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #FIQ_STACK_SIZE

    /* IRQ模式 */
    msr cpsr_c, #MODE_IRQ|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #IRQ_STACK_SIZE

    /* 系统模式 */
    msr cpsr_c, #MODE_SVC
    mov sp, r0

#ifdef __FLASH_BUILD__
    /* 如果是FLASH模式build，从ROM中复制数据段到RAM中 */
    ldr     r1, =_etext
    ldr     r2, =_data
    ldr     r3, =_edata
data_loop:
    cmp     r2, r3
    ldrlo   r0, [r1], #4
    strlo   r0, [r2], #4
    blo     data_loop
#else
    /* 重映射SRAM到零地址 */
    ldr r0, =0xFFFFFF00
    mov r1, #0x01
    str r1, [r0]
#endif

    /* 屏蔽所有IRQ中断 */
    ldr r1, =0xFFFFF124
    ldr r0, =0XFFFFFFFF
    str r0, [r1]
```

```
    ; 对bss段进行清零
    mov   r0,#0                        ; 置R0为0
    ldr   r1,=__bss_start             ; 获得bss段开始位置
    ldr   r2,=__bss_end               ; 获得bss段结束位置
bss_loop:
    cmp   r1,r2                        ; 确认是否已经到结束位置
    strlo r0,[r1],#4                   ; 清零
    blo   bss_loop                     ; 循环直到结束

    ; 对C++的全局对象进行构造
    ldr     r0, =__ctors_start__     ; 获得ctors开始位置
    ldr     r1, =__ctors_end__       ; 获得ctors结束位置
ctor_loop:
    cmp     r0, r1
    beq     ctor_end
    ldr     r2, [r0], #4
    stmfd   sp!, {r0-r1}
    mov     lr, pc
    bx              r2
    ldmfd   sp!, {r0-r1}
    b               ctor_loop
ctor_end:

    /* 跳转到RT-Thread Kernel */
    ldr pc, _rtthread_startup

_rtthread_startup: .word rtthread_startup

/* 异常处理 */
vector_undef: b vector_undef
vector_swi   : b vector_swi
vector_pabt : b vector_pabt
vector_dabt : b vector_dabt
vector_resv : b vector_resv

.globl rt_interrupt_enter
.globl rt_interrupt_leave
.globl rt_thread_switch_interrput_flag
.globl rt_interrupt_from_thread
.globl rt_interrupt_to_thread
/*
 * IRQ异常处理
 */
vector_irq:
    stmfd   sp!, {r0-r12,lr}              /* 先把R0 - R12，LR寄存器压栈保存 */
    bl      rt_interrupt_enter            /* 调用rt_interrupt_enter以确认进入中断处理 */
    bl      rt_hw_trap_irq                /* 调用C函数的中断处理函数进行处理 */
    bl      rt_interrupt_leave            /* 调用rt_interrupt_leave表示离开中断处理 */

    /* 如果设置了rt_thread_switch_interrput_flag，进行中断中的线程上下文处理 */
    ldr     r0, =rt_thread_switch_interrput_flag
    ldr     r1, [r0]
    cmp     r1, #1                        /* 判断是否设置了中断中线程切换标志 */
```

```
    beq     _interrupt_thread_switch        /* 是则跳转到_interrupt_thread_switch */

    ldmfd   sp!, {r0-r12,lr}                 /* R0 - R12，LR出栈 */
    subs    pc, lr, #4                       /* 中断返回 */

/*
 * FIQ异常处理
 * 在这里仅仅进行了简单的函数回调，OS并没对FIQ做特别处理。
 * 如果在FIQ中要用到OS的一些服务，需要做IRQ异常类似处理。
 */
vector_fiq:
    stmfd   sp!,{r0-r7,lr}                   /* R0 - R7，LR寄存器入栈，
                                              * FIQ模式下，R0 - R7是通用寄存器，
                                              * 其他的都是分组寄存器 */

    bl              rt_hw_trap_fiq          /* 跳转到rt_hw_trap_fiq进行处理 */
    ldmfd   sp!,{r0-r7,lr}
    subs            pc,lr,#4                 /* FIQ异常返回 */

/* 进行中断中的线程切换 */
_interrupt_thread_switch:
    mov             r1,  #0                  /* 清除切换标识 */
    str             r1,  [r0]

    ldmfd   sp!, {r0-r12,lr}                 /* 载入保存的R0 - R12及LR寄存器 */
    stmfd   sp!, {r0-r3}                     /* 先保存R0 - R3寄存器 */
    mov             r1,  sp                  /* 保存一份IRQ模式下的栈指针到R1寄存器 */
    add             sp,  sp, #16             /* IRQ栈中保持了R0 - R4，加16后刚好到栈底 */

    /* 后面会直接跳出IRQ模式，相当于恢复IRQ的栈 */

    sub             r2,  lr, #4              /* 保存中断前线程的PC到R2寄存器 */

    mrs             r3,  spsr                /* 保存中断前的CPSR到R3寄存器 */
    orr             r0,  r3, #NOINT          /* 关闭中断前线程的中断 */
    msr             spsr_c, r0

    ldr             r0,  =.+8                /* 把当前地址+8载入到R0寄存器中 */
    movs            pc,  r0                  /* 退出IRQ模式，由于SPSR被设置成关中断模式，
                                              * 所以从IRQ返回后，中断并没有打开

                                              * R0寄存器中的位置实际就是下一条指令，
                                              * 即PC继续往下走

                                              * 此时
                                              * 模式已经换成中断前的SVC模式，
                                              * SP寄存器也是SVC模式下的栈寄存器
                                              * R1保存IRQ模式下的栈指针
                                              * R2保存切换出线程的PC
                                              * R3保存切换出线程的CPSR */

    stmfd   sp!, {r2}                        /* 对R2寄存器压栈，即前面保存的切换出线程PC */
    stmfd   sp!, {r4-r12,lr}                 /* 对LR，R4 - R12寄存器进行压栈（切换出线程的）
```

```
    mov             r4,  r1                 /* R4寄存器为IRQ模式下的栈指针，
/* 栈中保存了切换出线程的R0 - R3 */

    mov             r5,  r3                 /* R5中保存了切换出线程的CPSR */
    ldmfd   r4!, {r0-r3}                    /* 恢复切换出线程的R0 - R3寄存器 */
    stmfd   sp!, {r0-r3}                    /* 对切换出线程的R0 - R3寄存器进行压栈 */
    stmfd   sp!, {r5}                       /* 对切换出线程的CPSR进行压栈 */
    mrs             r4,  spsr               /* 读取切换出线程的SPSR寄存器 */
    stmfd   sp!, {r4}                       /* 对切换出线程的SPSR进行压栈 */

    ldr     r4,  =rt_interrupt_from_thread
    ldr     r5,  [r4]
    str     sp,  [r5]                       /* 更新切换出线程的sp指针（存放在TCB中）*/

    ldr     r6,  =rt_interrupt_to_thread
    ldr     r6,  [r6]
    ldr     sp,  [r6]                       /* 获得切换到线程的栈指针 */

    ldmfd   sp!, {r4}                       /* 恢复切换到线程的SPSR寄存器 */
    msr     SPSR_cxsf, r4
    ldmfd   sp!, {r4}                       /* 恢复切换到线程的CPSR寄存器 */
    msr     CPSR_cxsf, r4

    ldmfd   sp!, {r0-r12,lr,pc}             /* 恢复切换到线程的R0 - R12，LR及PC寄存器 */
```

C.1.3 线程初始栈构造

在创建一个线程并把它放到就绪队列，调度器选择了这个线程开始运行时，调度器只知道要切换到这个线程中，它并不知道线程应该从什么地方开始运行，也不知道线程入口处应该放置哪些参数。为了解决这个问题，那么移植就需要"手工地"设置初始栈。添加一个stack.c文件以实现线程栈的初始化工作，代码如下。

```
#include <rtthread.h>
#define SVCMODE                 0x13

/**
 * This function will initialize thread stack
 *
 * @param tentry the entry of thread
 * @param parameter the parameter of entry
 * @param stack_addr the beginning stack address
 * @param texit the function will be called when thread exit
 *
 * @return stack address
 */
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter,
        rt_uint8_t *stack_addr, void *texit)
{
    unsigned long *stk;
```

```
    stk       = (unsigned long *)stack_addr;
    *(stk)    = (unsigned long)tentry;         /* 线程入口，等价于线程的PC */
    *(--stk) = (unsigned long)texit;           /* lr */
    *(--stk) = 0;                              /* r12 */
    *(--stk) = 0;                              /* r11 */
    *(--stk) = 0;                              /* r10 */
    *(--stk) = 0;                              /* r9 */
    *(--stk) = 0;                              /* r8 */
    *(--stk) = 0;                              /* r7 */
    *(--stk) = 0;                              /* r6 */
    *(--stk) = 0;                              /* r5 */
    *(--stk) = 0;                              /* r4 */
    *(--stk) = 0;                              /* r3 */
    *(--stk) = 0;                              /* r2 */
    *(--stk) = 0;                              /* r1 */
    *(--stk) = (unsigned long)parameter;       /* r0 : 入口函数参数 */
    *(--stk) = SVCMODE;                        /* cpsr，采用SVC模式运行 */
    *(--stk) = SVCMODE;                        /* spsr */

    /* return task's current stack address */
    return (rt_uint8_t *)stk;
}
```

C.1.4 中断处理

当一个中断触发时，从上面初始化代码中可以看到，它的处理流程是这样的：

rt_hw_trap_irq是实际的中断服务例程调用函数，它在trap.c文件中实现。

```
#include <rtthread.h>
#include <rthw.h>

#include "AT91SAM7S.h"

/* 实际的中断处理函数 */
void rt_hw_trap_irq()
{
    /* 从IVR寄存器中获得当前设定的中断服务例程函数入口 */
    rt_isr_handler_t hander = (rt_isr_handler_t)AT91C_AIC_IVR;

    /* 调用中断服务例程函数，ISR寄存器指示出是第几号中断 */
    hander(AT91C_AIC_ISR);

    /* 写EOICR寄存器以指示出中断服务结束 */
    AT91C_AIC_EOICR = 0;
}

/* FIQ异常处理函数，目前未使用到 */
void rt_hw_trap_fiq()
{
    rt_kprintf("fast interrupt request\n");
}
```

中断触发

```
/* 异常向量表，通常位于0地址 */
b       reset
ldr     pc, _vector_undef
ldr     pc, _vector_swi
ldr     pc, _vector_pabt
ldr     pc, _vector_dabt
nop
ldr     pc, _vector_irq
ldr     pc, _vector_fiq
```

```
vector_irq:
  stmfd sp!, {r0-r12,lr}
  bl    rt_interrupt_enter
  bl    rt_hw_trap_irq
  bl    rt_interrupt_leave

  /* 中断中的任务切换处理 */
  ldr   r0, =rt_thread_switch_interrput_flag
  ldr   r1, [r0]
  cmp r1, #1
  beq  rt_hw_context_switch_interrupt_do

  ldmfd sp!, {r0-r12,lr}
  subspc, lr, #4
```

trap.c::rt_hw_trap_irq

Figure C.2: 启动文件中的汇编处理

在rt_hw_trap_irq函数中，它只是负责找到当前产生的中断应该调用哪个中断服务例程，而并没给出如何去设置每个中断所对应的中断服务例程。中断控制器及中断服务例程的设定在interrupt.c中实现。

```
#include <rtthread.h>
#include "AT91SAM7S.h"

/* 总共32个中断 */
#define MAX_HANDLERS    32

extern rt_uint32_t rt_interrupt_nest;

rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;
rt_uint32_t rt_thread_switch_interrput_flag;

/* 默认的中断处理 */
void rt_hw_interrupt_handler(int vector)
{
    rt_kprintf("Unhandled interrupt %d occured!!!\n", vector);
}

/* 初始化中断控制器 */
void rt_hw_interrupt_init()
{
    rt_base_t index;

    /* 每个中断服务例程都设置到默认的中断处理上 */
    for (index = 0; index < MAX_HANDLERS; index ++)
    {
        AT91C_AIC_SVR(index) = (rt_uint32_t)rt_hw_interrupt_handler;
    }

    /* 初始化线程在中断中切换的一些变量 */
    rt_interrupt_nest = 0;
    rt_interrupt_from_thread = 0;
    rt_interrupt_to_thread = 0;
    rt_thread_switch_interrput_flag = 0;
}

/* 屏蔽某个中断的API */
void rt_hw_interrupt_mask(int vector)
{
    /* disable interrupt */
    AT91C_AIC_IDCR = 1 << vector;

    /* clear interrupt */
    AT91C_AIC_ICCR = 1 << vector;
}

/* 去屏蔽某个中断的API */
void rt_hw_interrupt_umask(int vector)
{
    AT91C_AIC_IECR = 1 << vector;
```

```
}

/* 在相应的中断号上装载中断服务例程 */
void rt_hw_interrupt_install(int vector, rt_isr_handler_t new_handler, rt_isr_handler_t *old_handler)
{
    if(vector >= 0 && vector < MAX_HANDLERS)
    {
        if (*old_handler != RT_NULL) *old_handler = (rt_isr_handler_t)AT91C_AIC_SVR(vector);
        if (new_handler != RT_NULL) AT91C_AIC_SVR(vector) = (rt_uint32_t)new_handler;
    }
}
```

C.1.5 串口设备驱动

为了能够使用finsh shell，系统中必须实现一个相应的设备，这里实现了一个基本的串口类设备（文件: serial.c），详细的代码解释请参考 *AT91SAM7S64*串口驱动 。

C.2 板级相关移植

bsp目录下放置了各类开发板/平台的具体实现，包括开发板/平台的初始化，外设的驱动等。由于AT91SAM7S64核心板中并没特别外扩设备，所以这个目录中只包含了相应的初始化即可。

Tip: 通常SoC芯片已经集成了很多外设，所以这个目录中文件相对比较少，比较简单，而这些SoC芯片的外设驱动则放置在CPU相关的实现中，以方便不同的开发板复用。

C.2.1 配置头文件

RT-Thread代码中默认包含rtconfig.h头文件作为它的配置文件，会定义RT-Thread中各种选项设置，例如内核对象名称长度，是否支持线程间通信中的信箱，消息队列，快速事件等。详细情况请参看 内核配置_ ，和此移植相关的配置文件代码如下：

```
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* 系统中对象名称大小 */
#define RT_NAME_MAX             4

/* 对齐方式 */
#define RT_ALIGN_SIZE           4

/* 最大支持的优先级：32 */
#define RT_THREAD_PRIORITY_MAX  32

/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100

/* SECTION: 调试选项 */
/* 调试 */
```

```
/* #define RT_THREAD_DEBUG */

/* 支持线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE

/* 支持互斥锁 */
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持快速事件 */
/* #define RT_USING_FASTEVENT */

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
#define RT_CONSOLEBUF_SIZE  128

/* SECTION: FinSH shell 选项 */
```

```
/* 支持finsh作为shell */
#define RT_USING_FINSH
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
/* 使用小型libc库 */
#define RT_USING_MINILIBC

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

#endif
```

由于采用了GNU GCC作为编译器，并没有编译代码的限制，所以这里打开了finsh shell组件、C++支持组件。另外由于GNU GCC并不存在C库，所以在配置文件中使能了小型C库支持。

C.2.2 Kernel启动

做为系统启动汇编后进入的第一个C函数，startup.c文件实现了相应的rtthread_startup函数（针对RealView MDK，则实现了main函数）。

```
/* 为了获得heap的起始地址，把bss段的末地址通过链接器定义的方式给出 */
#ifdef __CC_ARM
extern int Image$$RW_IRAM1$$ZI$$Limit;
#endif

#ifdef __GNUC__
extern unsigned char __bss_start;
extern unsigned char __bss_end;
#endif

extern void rt_hw_interrupt_init(void);
extern int  rt_application_init(void);
#ifdef RT_USING_DEVICE
extern rt_err_t rt_hw_serial_init(void);
#endif

/**
 * RT-Thread启动函数
 */
void rtthread_startup(void)
{
    /* 初始化中断 */
    rt_hw_interrupt_init();

    /* 初始化开发板硬件 */
    rt_hw_board_init();

    /* 显示RT-Thread版本信息 */
```

```
    rt_show_version();

    /* 初始化系统节拍 */
    rt_system_tick_init();

    /* 初始化内核对象 */
    rt_system_object_init();

    /* 初始化系统定时器 */
    rt_system_timer_init();

    /* 初始化堆内存，__bss_end在链接脚本中定义 */
#ifdef RT_USING_HEAP
#ifdef __CC_ARM
    rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)0x204000);
#elif __ICCARM__
    rt_system_heap_init(__segment_end("HEAP"), (void*)0x204000);
#else
    rt_system_heap_init((void*)&__bss_end, (void*)0x204000);
#endif
#endif

    /* 初始化系统调度器 */
    rt_system_scheduler_init();

#ifdef RT_USING_HOOK
    /* 设置空闲线程的钩子函数 */
    rt_thread_idle_sethook(rt_hw_led_flash);
#endif

#ifdef RT_USING_DEVICE
    /* init hardware serial device */
    rt_hw_serial_init();
    /* init all device */
    rt_device_init_all();
#endif

    /* 初始化用户程序 */
    rt_application_init();

#ifdef RT_USING_FINSH
    /* init finsh */
    finsh_system_init();
    finsh_set_device("uart1");
#endif

    /* 初始化IDLE线程 */
    rt_thread_idle_init();

    /* 开始启动系统调度器，切换到第一个线程中 */
    rt_system_scheduler_start();

    /* 此处应该是永远不会达到的 */
```

```
    return ;
}

int main (void)
{
    /* 在main函数中调用rtthread_startup函数 */
    rtthread_startup();

    return 0;
}
```

C.2.3 开发板初始化

和开发板硬件相关的实现放于board.c文件中，代码及注释如下：

```
/* Periodic Interval Value */
#define PIV  (((MCK/16)/1000)*(1000/RT_TICK_PER_SECOND))

/* OS时钟定时器中断服务例程，这个是移植中必须添加的部分 */
void rt_hw_timer_handler(int vector)
{
    if (AT91C_PITC_PISR & 0x01)
    {
        /* 定时器中断到了，调用rt_tick_increase通知OS一个时钟节拍达到 */
        rt_tick_increase();

        /* 确认中断结束 */
        AT91C_AIC_EOICR = AT91C_PITC_PIVR;
    }
    else
    {
        /* 确认中断结束 */
        AT91C_AIC_EOICR = 0;
    }
}

/* AT91SAM7S64核心板上PIO8连接了一个LED灯 */
                        /* PIO   Flash    PA    PB   PIN */
#define LED     (1 << 8)/* PA8          & TWD  NPCS3  43 */

/* 开发板中LED的初始化化 */
static void rt_hw_board_led_init()
{
    /* 使能PIO的时钟 */
    AT91C_PMC_PCER = 1 << AT91C_ID_PIOA;

    /* 配置PIO8为输出 */
    AT91C_PIO_PER = LED;
    AT91C_PIO_OER = LED;
}

/* 点亮LED灯 */
```

```
void rt_hw_board_led_on()
{
    AT91C_PIO_CODR = LED;
}

/* 熄灭LED灯 */
void rt_hw_board_led_off()
{
    AT91C_PIO_SODR = LED;
}

/* 采用循环延时的方式对LED进行闪烁 */
void rt_hw_led_flash()
{
    int i;

    rt_hw_board_led_off();
    for (i = 0; i < 2000000; i ++);

    rt_hw_board_led_on();
    for (i = 0; i < 2000000; i ++);
}

/*
 * RT-Thread Console 接口，由rt_kprintf使用
 */
/* 在console上显示一段字符串 str，它不应触发一个硬件中断 */
void rt_hw_console_output(const char* str)
{
    while (*str)
    {
        /* 如果是'\n'，在前面插入一个'\r'。 */
        if (*str == '\n')
        {
            while (!(AT91C_US0_CSR & AT91C_US_TXRDY));
            AT91C_US0_THR = '\r';
        }

        /* 等待发送空 */
        while (!(AT91C_US0_CSR & AT91C_US_TXRDY));

        /* 发送一个字符 */
        AT91C_US0_THR = *str;

        /* 移动显示字符串指针到下一个位置 */
        str ++;
    }
}

/* console的初始化函数 */
static void rt_hw_console_init()
{
    /* 使能USART0时钟 */
```

```
    AT91C_PMC_PCER = 1 << AT91C_ID_US0;
    /* 设置相应的接收、发送Pin脚 */
    AT91C_PIO_PDR = (1 << 5) | (1 << 6);

    /* 先重置控制器 */
    AT91C_US0_CR = AT91C_US_RSTRX    |      /* Reset Receiver      */
                   AT91C_US_RSTTX    |      /* Reset Transmitter   */
                   AT91C_US_RXDIS    |      /* Receiver Disable    */
                   AT91C_US_TXDIS;          /* Transmitter Disable */

    /* 初始化控制器 */
    AT91C_US0_MR = AT91C_US_USMODE_NORMAL |    /* Normal Mode */
                   AT91C_US_CLKS_CLOCK    |    /* Clock = MCK */
                   AT91C_US_CHRL_8_BITS   |    /* 8-bit Data  */
                   AT91C_US_PAR_NONE      |    /* No Parity   */
                   AT91C_US_NBSTOP_1_BIT;      /* 1 Stop Bit  */

    /* 设置波特率 */
    AT91C_US0_BRGR = BRD;

    /* 使能接收和发送 */
    AT91C_US0_CR = AT91C_US_RXEN |  /* Receiver Enable     */
                   AT91C_US_TXEN;   /* Transmitter Enable  */
}

/* AT91SAM7S64核心板初始化 */
void rt_hw_board_init()
{
    /* 初始化console，在使用rt_kprintf前，必须先要初始化console */
    rt_hw_console_init();

    /* 初始化LED */
    rt_hw_board_led_init();

    /* 初始化PITC */
    AT91C_PITC_PIMR = (1 << 25) | (1 << 24) | PIV;
    /* 装置OS定时器中断服务例程 */
    rt_hw_interrupt_install(AT91C_ID_SYS, rt_hw_timer_handler, RT_NULL);
    AT91C_AIC_SMR(AT91C_ID_SYS) = 0;
    rt_hw_interrupt_umask(AT91C_ID_SYS);
}
```

C.2.4 用户初始化文件

作为基本的移植，只需要一个空的rt_application_init函数实现即可。

```
/* 只建立一个空内核，直接返回即可 */
int rt_application_init()
{
    return 0;   /* empty */
}
```

C.2.5 链接脚本

由于AT91SAM7S64只包含较小的内存，所以才有直接在片内flash中运行的方式，在文件sam7s_rom.lds中实现。

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
MEMORY
{
    CODE (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000
    DATA (rw) : ORIGIN = 0x00200000, LENGTH = 0x00004000
}
ENTRY(_start)
SECTIONS
{
    .text :
    {
        *(.init)
        *(.text)
    } > CODE = 0

    . = ALIGN(4);
    .rodata :
    {
        *(.rodata .rodata.*)
    } > CODE

    _etext = . ;
    PROVIDE (etext = .);

    /* .data section which is used for initialized data */

    .data : AT (_etext)
    {
        _data = . ;
        *(.data)
        SORT(CONSTRUCTORS)
    } >DATA
    . = ALIGN(4);

    _edata = . ;
    PROVIDE (edata = .);

    . = ALIGN(4);
    __bss_start = .;
    .bss :
    {
        *(.bss)
    } > DATA
    __bss_end = .;

    _end = .;
}
```

通过这个链接脚本文件，主要生成了几个section：

1. .text，从0x00000000开始，放置可执行代码部分。
2. .rodata，紧接着.text后面放置，其中包含了只读数据；并在后面插入了_etext符合，指向结束地址。
3. .data，在映像文件中放置于_etext位置，其中包含了可读写的数据，但在运行状态下则从0x00200000地址开始。
4. .bss，紧接着.data放置，并在开始位置及结束位置放置了__bss_start和__bss_end以指向相应的位置。

APPENDIX

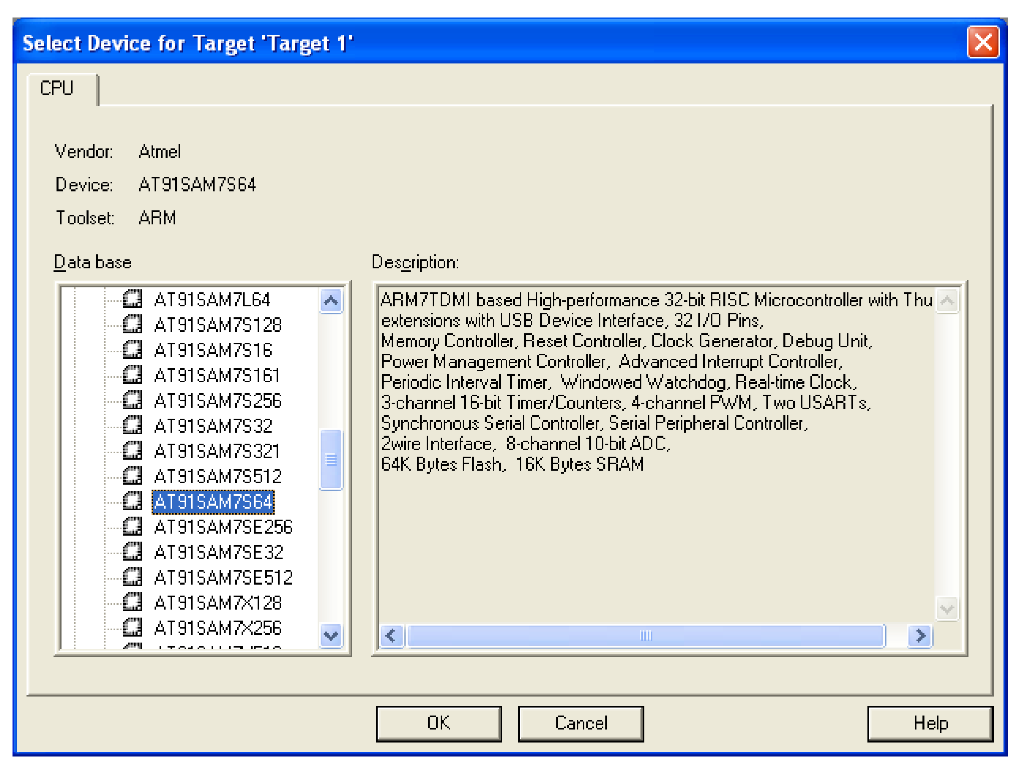
D

REALVIEW MDK移植

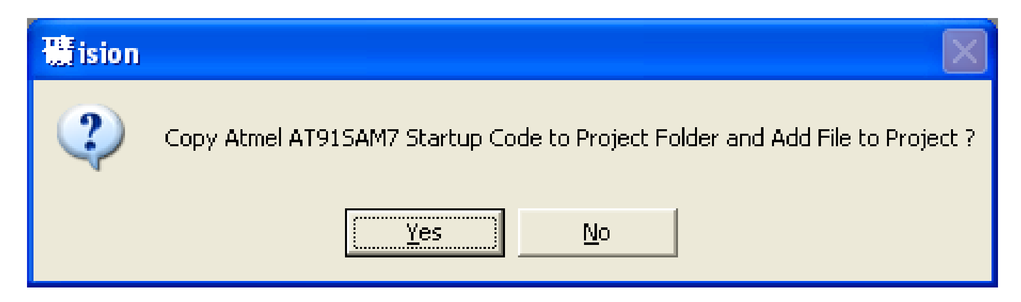
本节用到的RealView MDK版本是3.50评估版，因为生成的代码小于16k，可以正常编译调试。（由于RealView MDK评估版和RealView MDK专业版的差异，专业版会生成更小的代码尺寸，推荐使用专业版）

D.1 建立RealView MDK工程

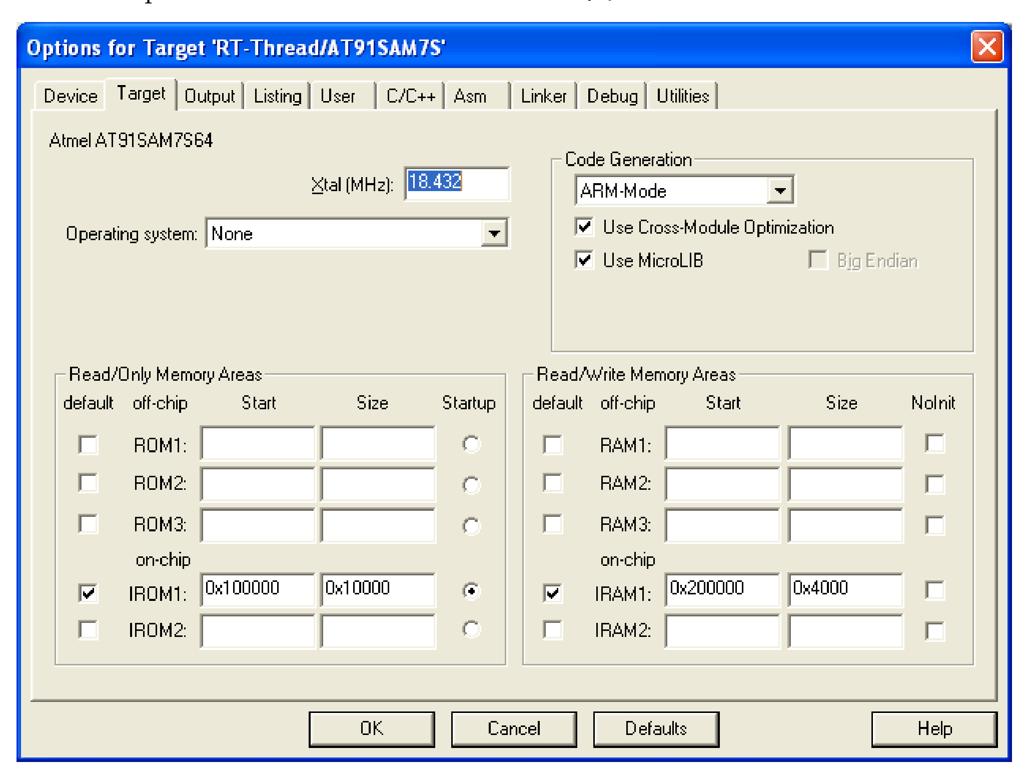
在kernel/bsp目录下新建sam7s目录。在RealView MDK中新建立一个工程文件（用菜单创建），名称为project，保存在kernel/bsp/sam7s目录下。创建工程时一次的选项如下：CPU选择Atmel的AT91SAM7S64



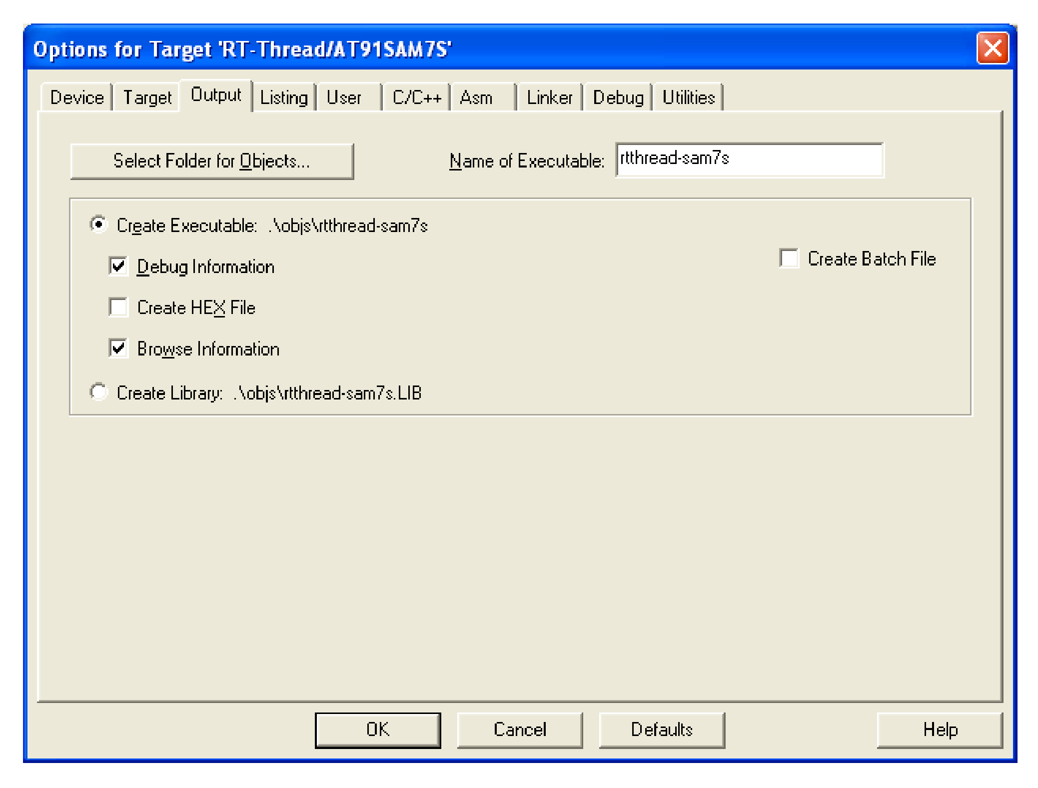
提问复制Atmel AT91SAM7S的启动代码到工程目录，确认 Yes



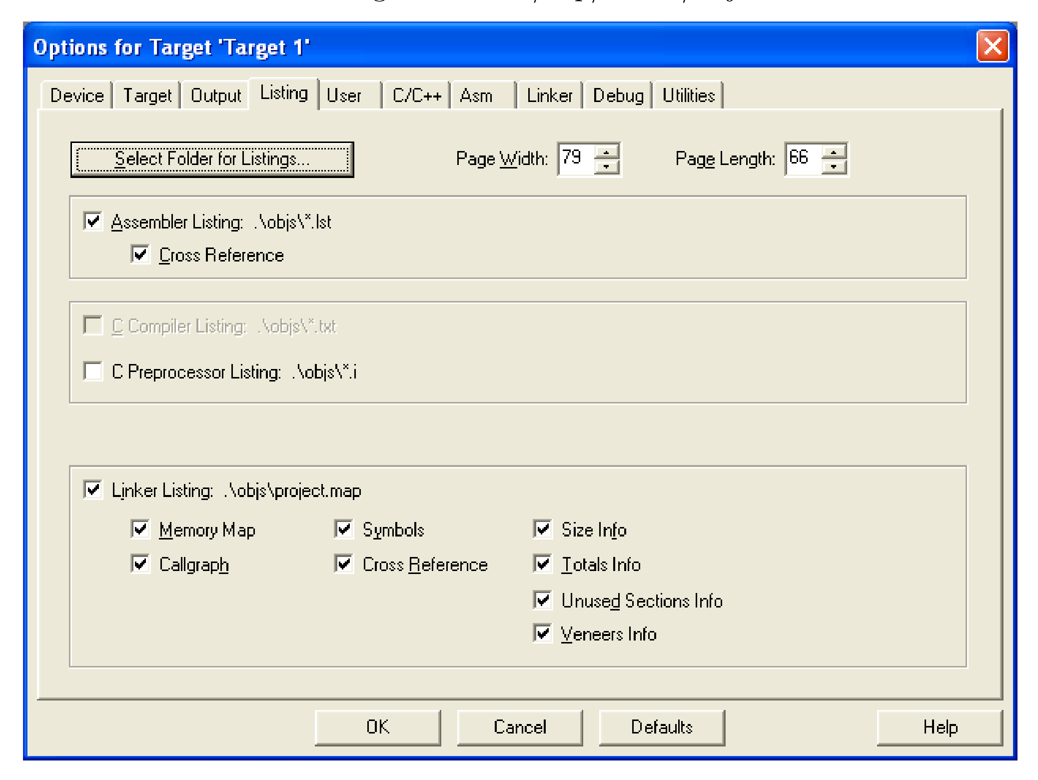
然后选择工程的属性，Code Generation选择ARM-Mode，如果希望产生更小的代码选择Use Cross-Module Optimization和Use MicroLIB，如下图



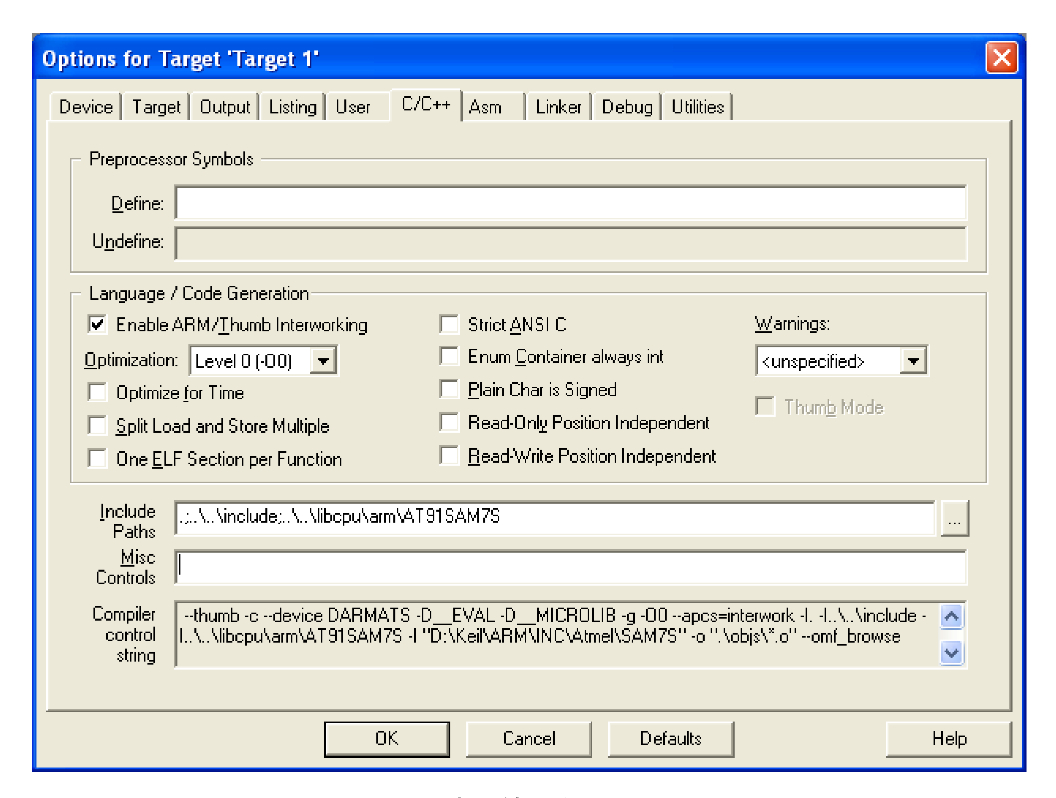
Select Folder for Objects目录选择到kernel/bsp/sam7s/objs， Name of Executable为rtthread-sam7s。



同样Select Folder for Listings选择kernel/bsp/sam7s/objs目录，如下图所示：



C/C++编译选项标签页中，选择Enable ARM/Thumb Interworking，Include Paths（头文件搜索路径）中添加上目录kernel\include，kernel\libcpu\arm\AT91SAM7S以及kernel\bsp\sam7s目录，如下图所示：



Asm，Linker，Debug和Utilities选项使用初始配置。

D.2 添加RT-Thread的源文件

对工程中初始添加的Source Group1改名为Startup，并添加Kernel，AT91SAM7S的Group，开始建立工程时产生的SAM7.s重命名为start_rvds.s并放到kernellibcpuAT91SAM7S目录中。

Kernel Group中添加所有kernel\src下的C源文件； Startup Group中添加startup.c，board.c文件（放于kernel\bsp\sam7s目录中）； AT91SAM7S Group中添加context_rvds.s，stack.c，trap.c，interrupt.c等文件（放于kernellibcpusam7s目录中）；

在kernel/bsp/sam7s目录中添加rtconfig.h文件，内容如下：

```
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* 系统中对象名称大小 */
#define RT_NAME_MAX             4

/* 对齐方式 */
#define RT_ALIGN_SIZE           4

/* 最大支持的优先级：32 */
#define RT_THREAD_PRIORITY_MAX  32

/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100
```

```
/* SECTION: 调试选项 */
/* 调试 */
/* #define RT_THREAD_DEBUG */

/* 支持线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE

/* 支持互斥锁 */
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持快速事件 */
/* #define RT_USING_FASTEVENT */

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
```

```
#define RT_CONSOLEBUF_SIZE  128

/* SECTION: FinSH shell 选项 */
/* 支持finsh作为shell */
/* #define RT_USING_FINSH */
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
/* 使用小型libc库 */
/* #define RT_USING_MINILIBC */

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

#endif
```

由于采用的是RealView MDK评估版本，加入finsh shell将超出代码限制，所以此处把RT_USING_FINSH的宏定义移除了。

D.3 线程上下文切换

代码 A - 17 context_rvds.s

```
NOINT   EQU     0xc0    ; disable interrupt in psr

    AREA |.text|, CODE, READONLY, ALIGN=2
    ARM
    REQUIRE8
    PRESERVE8

; rt_base_t rt_hw_interrupt_disable();
; 关闭中断，关闭前返回CPSR寄存器值
rt_hw_interrupt_disable PROC
    EXPORT rt_hw_interrupt_disable
    MRS r0, cpsr
    ORR r1, r0, #NOINT
    MSR cpsr_c, r1
    BX  lr
    ENDP

; void rt_hw_interrupt_enable(rt_base_t level);
; 恢复中断状态
rt_hw_interrupt_enable  PROC
    EXPORT rt_hw_interrupt_enable
    MSR cpsr_c, r0
    BX  lr
    ENDP
```

```
; void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; r0 --> from
; r1 --> to
; 进行线程的上下文切换
rt_hw_context_switch    PROC
    EXPORT rt_hw_context_switch
    STMFD   sp!, {lr}             ; 把LR寄存器压入栈（这个函数返回后的下一个执行处）
    STMFD   sp!, {r0-r12, lr}     ; 把R0 - R12以及LR压入栈

    MRS     r4, cpsr              ; 读取CPSR寄存器到R4寄存器
    STMFD   sp!, {r4}             ; 把R4寄存器压栈（即上一指令取出的CPSR寄存器）
    MRS     r4, spsr              ; 读取SPSR寄存器到R4寄存器
    STMFD   sp!, {r4}             ; 把R4寄存器压栈（即SPSR寄存器）

    STR sp, [r0]                  ; 把栈指针更新到TCB的sp，是由R0传入此函数

    ; 到这里换出线程的上下文都保存在栈中

    LDR sp, [r1]                  ; 载入切换到线程的TCB的sp

    ; 从切换到线程的栈中恢复上下文，次序和保存的时候刚好相反

    LDMFD   sp!, {r4}             ; 出栈到R4寄存器（保存了SPSR寄存器）
    MSR spsr_cxsf, r4            ; 恢复SPSR寄存器
    LDMFD   sp!, {r4}             ; 出栈到R4寄存器（保存了CPSR寄存器）
    MSR cpsr_cxsf, r4            ; 恢复CPSR寄存器

    LDMFD   sp!, {r0-r12, lr, pc}   ; 对R0 - R12及LR、PC进行恢复
    ENDP

; void rt_hw_context_switch_to(rt_uint32 to);
; r0 --> to
; 此函数只在系统进行第一次发生任务切换时使用，因为是从没有线程的状态进行切换
; 实现上，刚好是rt_hw_context_switch的下半截
rt_hw_context_switch_to PROC
    EXPORT rt_hw_context_switch_to
    LDR sp, [r0]                  ; 获得切换到线程的SP指针

    LDMFD   sp!, {r4}             ; 出栈R4寄存器（保存了SPSR寄存器值）
    MSR spsr_cxsf, r4            ; 恢复SPSR寄存器
    LDMFD   sp!, {r4}             ; 出栈R4寄存器（保存了CPSR寄存器值）
    MSR cpsr_cxsf, r4            ; 恢复CPSR寄存器

    LDMFD   sp!, {r0-r12, lr, pc}   ; 恢复R0 - R12，LR及PC寄存器
    ENDP

    IMPORT rt_thread_switch_interrput_flag
    IMPORT rt_interrupt_from_thread
    IMPORT rt_interrupt_to_thread

; void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
; 此函数会在调度器中调用，在调度器做上下文切换前会判断是否处于中断服务模式中，如果
; 是则调用rt_hw_context_switch_interrupt函数（设置中断中任务切换标志）
```

```
; 否则调用 rt_hw_context_switch函数（进行真正的线程上线文切换）
rt_hw_context_switch_interrupt  PROC
    EXPORT rt_hw_context_switch_interrupt
    LDR r2, =rt_thread_switch_interrput_flag
    LDR r3, [r2]                    ; 载入中断中切换标致地址
    CMP r3, #1                      ; 等于 1 ?
    BEQ _reswitch                   ; 如果等于1，跳转到_reswitch
    MOV r3, #1                      ; 设置中断中切换标志位1
    STR r3, [r2]                    ; 保存到标志变量中
    LDR r2, =rt_interrupt_from_thread
    STR r0, [r2]                    ; 保存切换出线程栈指针
_reswitch
    LDR r2, =rt_interrupt_to_thread
    STR r1, [r2]                    ; 保存切换到线程栈指针
    BX  lr
    ENDP

    END
```

D.4 启动汇编文件

启动汇编文件可直接在RealView MDK新创建的SAM7.s文件上进行修改得到，把它重命名（为了和RT-Thread的文件命名规则保持一致）为start_rvds.s。修改主要有几点： 默认IRQ中断是由RealView的库自己处理的，RT-Thread需要截获下来进行做操作系统级的调度； 自动生成的SAM7.s默认对Watch Dog不做处理，修改成disable状态（否则需要在代码中加入相应代码）； 在汇编文件最后跳转到RealView的库函数__main时，会提前转到ARM的用户模式，RT-Thread需要维持在SVC模式； 和GNU GCC的移植类似，需要添加中断结束后的线程上下文切换部分代码。

代码A-8是启动汇编的代码清单，其中加双下划线部分是修改的部分。代码 A - 18 start_rvds.s

```
;/*****************************************************************************/
;/* SAM7.S: Startup file for Atmel AT91SAM7 device series                     */
;/*****************************************************************************/
;/* <<< Use Configuration Wizard in Context Menu >>>                          */
;/*****************************************************************************/
;/* This file is part of the uVision/ARM development tools.                   */
;/* Copyright (c) 2005-2006 Keil Software. All rights reserved.               */
;/* This software may only be used under the terms of a valid, current,       */
;/* end user licence from KEIL for a compatible version of KEIL software      */
;/* development tools. Nothing else gives you the right to use this software. */
;/*****************************************************************************/

;/*
; *  The SAM7.S code is executed after CPU Reset. This file may be
; *  translated with the following SET symbols. In uVision these SET
; *  symbols are entered under Options - ASM - Define.
; *
; *  REMAP: when set the startup code remaps exception vectors from
; *  on-chip RAM to address 0.
```

```
; *
; *  RAM_INTVEC: when set the startup code copies exception vectors
; *  from on-chip Flash to on-chip RAM.
; */


; Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs

Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UND        EQU     0x1B
Mode_SYS        EQU     0x1F

I_Bit           EQU     0x80            ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40            ; when F bit is set, FIQ is disabled


; Internal Memory Base Addresses
FLASH_BASE      EQU     0x00100000
RAM_BASE        EQU     0x00200000


;// <h> Stack Configuration (Stack Sizes in Bytes)
;//   <o0> Undefined Mode      <0x0-0xFFFFFFFF:8>
;//   <o1> Supervisor Mode     <0x0-0xFFFFFFFF:8>
;//   <o2> Abort Mode          <0x0-0xFFFFFFFF:8>
;//   <o3> Fast Interrupt Mode <0x0-0xFFFFFFFF:8>
;//   <o4> Interrupt Mode      <0x0-0xFFFFFFFF:8>
;//   <o5> User/System Mode    <0x0-0xFFFFFFFF:8>
;// </h>

UND_Stack_Size  EQU     0x00000000
SVC_Stack_Size  EQU     0x00000080
ABT_Stack_Size  EQU     0x00000000
FIQ_Stack_Size  EQU     0x00000000
IRQ_Stack_Size  EQU     0x00000080
USR_Stack_Size  EQU     0x00000400

ISR_Stack_Size  EQU     (UND_Stack_Size + SVC_Stack_Size + ABT_Stack_Size + \
                         FIQ_Stack_Size + IRQ_Stack_Size)

AREA    STACK, NOINIT, READWRITE, ALIGN=3

Stack_Mem       SPACE   USR_Stack_Size
__initial_sp    SPACE   ISR_Stack_Size
Stack_Top


;// <h> Heap Configuration
;//   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF>
```

```
;// </h>

Heap_Size       EQU     0x00000000

AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit


; Reset Controller (RSTC) definitions
RSTC_BASE       EQU     0xFFFFFD00      ; RSTC Base Address
RSTC_MR         EQU     0x08            ; RSTC_MR Offset

;/*
;// <e> Reset Controller (RSTC)
;//   <o1.0>     URSTEN: User Reset Enable
;//              <i> Enables NRST Pin to generate Reset
;//   <o1.8..11> ERSTL: External Reset Length <0-15>
;//              <i> External Reset Time in 2^(ERSTL+1) Slow Clock Cycles
;// </e>
;*/
RSTC_SETUP      EQU     1
RSTC_MR_Val     EQU     0xA5000401


; Embedded Flash Controller (EFC) definitions
EFC_BASE        EQU     0xFFFFFF00      ; EFC Base Address
EFC0_FMR        EQU     0x60            ; EFC0_FMR Offset
EFC1_FMR        EQU     0x70            ; EFC1_FMR Offset

;// <e> Embedded Flash Controller 0 (EFC0)
;//   <o1.16..23> FMCN: Flash Microsecond Cycle Number <0-255>
;//               <i> Number of Master Clock Cycles in 1us
;//   <o1.8..9>    FWS: Flash Wait State
;//               <0=> Read: 1 cycle / Write: 2 cycles
;//               <1=> Read: 2 cycle / Write: 3 cycles
;//               <2=> Read: 3 cycle / Write: 4 cycles
;//               <3=> Read: 4 cycle / Write: 4 cycles
;// </e>
EFC0_SETUP      EQU     1
EFC0_FMR_Val    EQU     0x00320100

;// <e> Embedded Flash Controller 1 (EFC1)
;//   <o1.16..23> FMCN: Flash Microsecond Cycle Number <0-255>
;//               <i> Number of Master Clock Cycles in 1us
;//   <o1.8..9>    FWS: Flash Wait State
;//               <0=> Read: 1 cycle / Write: 2 cycles
;//               <1=> Read: 2 cycle / Write: 3 cycles
;//               <2=> Read: 3 cycle / Write: 4 cycles
;//               <3=> Read: 4 cycle / Write: 4 cycles
;// </e>
EFC1_SETUP      EQU     0
```

```
EFC1_FMR_Val    EQU     0x00320100


; Watchdog Timer (WDT) definitions
WDT_BASE        EQU     0xFFFFFD40      ; WDT Base Address
WDT_MR          EQU     0x04            ; WDT_MR Offset

;// <e> Watchdog Timer (WDT)
;//   <o1.0..11>  WDV: Watchdog Counter Value <0-4095>
;//   <o1.16..27> WDD: Watchdog Delta Value <0-4095>
;//   <o1.12>     WDFIEN: Watchdog Fault Interrupt Enable
;//   <o1.13>     WDRSTEN: Watchdog Reset Enable
;//   <o1.14>     WDRPROC: Watchdog Reset Processor
;//   <o1.28>     WDDBGHLT: Watchdog Debug Halt
;//   <o1.29>     WDIDLEHLT: Watchdog Idle Halt
;//   <o1.15>     WDDIS: Watchdog Disable
;// </e>
WDT_SETUP       EQU     1
WDT_MR_Val      EQU     0x00008000


; Power Mangement Controller (PMC) definitions
PMC_BASE        EQU     0xFFFFFC00      ; PMC Base Address
PMC_MOR         EQU     0x20            ; PMC_MOR Offset
PMC_MCFR        EQU     0x24            ; PMC_MCFR Offset
PMC_PLLR        EQU     0x2C            ; PMC_PLLR Offset
PMC_MCKR        EQU     0x30            ; PMC_MCKR Offset
PMC_SR          EQU     0x68            ; PMC_SR Offset
PMC_MOSCEN      EQU     (1<<0)          ; Main Oscillator Enable
PMC_OSCBYPASS   EQU     (1<<1)          ; Main Oscillator Bypass
PMC_OSCOUNT     EQU     (0xFF<<8)       ; Main OScillator Start-up Time
PMC_DIV         EQU     (0xFF<<0)       ; PLL Divider
PMC_PLLCOUNT    EQU     (0x3F<<8)       ; PLL Lock Counter
PMC_OUT         EQU     (0x03<<14)      ; PLL Clock Frequency Range
PMC_MUL         EQU     (0x7FF<<16)     ; PLL Multiplier
PMC_USBDIV      EQU     (0x03<<28)      ; USB Clock Divider
PMC_CSS         EQU     (3<<0)          ; Clock Source Selection
PMC_PRES        EQU     (7<<2)          ; Prescaler Selection
PMC_MOSCS       EQU     (1<<0)          ; Main Oscillator Stable
PMC_LOCK        EQU     (1<<2)          ; PLL Lock Status
PMC_MCKRDY      EQU     (1<<3)          ; Master Clock Status

;// <e> Power Mangement Controller (PMC)
;//   <h> Main Oscillator
;//     <o1.0>      MOSCEN: Main Oscillator Enable
;//     <o1.1>      OSCBYPASS: Oscillator Bypass
;//     <o1.8..15>  OSCCOUNT: Main Oscillator Startup Time <0-255>
;//   </h>
;//   <h> Phase Locked Loop (PLL)
;//     <o2.0..7>   DIV: PLL Divider <0-255>
;//     <o2.16..26> MUL: PLL Multiplier <0-2047>
;//                 <i> PLL Output is multiplied by MUL+1
;//     <o2.14..15> OUT: PLL Clock Frequency Range
```

```
;//                 <0=> 80..160MHz  <1=> Reserved
;//                 <2=> 150..220MHz <3=> Reserved
;//     <o2.8..13>  PLLCOUNT: PLL Lock Counter <0-63>
;//     <o2.28..29> USBDIV: USB Clock Divider
;//                 <0=> None  <1=> 2  <2=> 4  <3=> Reserved
;//   </h>
;//   <o3.0..1>   CSS: Clock Source Selection
;//               <0=> Slow Clock
;//               <1=> Main Clock
;//               <2=> Reserved
;//               <3=> PLL Clock
;//   <o3.2..4>   PRES: Prescaler
;//               <0=> None
;//               <1=> Clock / 2    <2=> Clock / 4
;//               <3=> Clock / 8    <4=> Clock / 16
;//               <5=> Clock / 32   <6=> Clock / 64
;//               <7=> Reserved
;// </e>
PMC_SETUP       EQU     1
PMC_MOR_Val     EQU     0x00000601
PMC_PLLR_Val    EQU     0x00191C05
PMC_MCKR_Val    EQU     0x00000007

        PRESERVE8

; Area Definition and Entry Point
;  Startup Code must be linked first at Address at which it expects to run.

AREA    RESET, CODE, READONLY
ARM


; Exception Vectors
;  Mapped to Address 0.
;  Absolute addressing mode must be used.
;  Dummy Handlers are implemented as infinite loops which can be modified.

Vectors         LDR     PC,Reset_Addr
                LDR     PC,Undef_Addr
                LDR     PC,SWI_Addr
                LDR     PC,PAbt_Addr
                LDR     PC,DAbt_Addr
                NOP                             ; Reserved Vector
                LDR     PC,IRQ_Addr
                LDR     PC,FIQ_Addr

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
        DCD     0                   ; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
```

```
FIQ_Addr        DCD     FIQ_Handler

Undef_Handler   B       Undef_Handler
SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler

; IRQ和FIQ的处理由操作系统截获，需要重新实现
; IRQ_Handler   B       IRQ_Handler

FIQ_Handler     B       FIQ_Handler

; Reset Handler

                EXPORT  Reset_Handler
Reset_Handler

; Setup RSTC
                IF      RSTC_SETUP != 0
                LDR     R0, =RSTC_BASE
                LDR     R1, =RSTC_MR_Val
                STR     R1, [R0, #RSTC_MR]
                ENDIF


; Setup EFC0
                IF      EFC0_SETUP != 0
                LDR     R0, =EFC_BASE
                LDR     R1, =EFC0_FMR_Val
                STR     R1, [R0, #EFC0_FMR]
                ENDIF

; Setup EFC1
                IF      EFC1_SETUP != 0
                LDR     R0, =EFC_BASE
                LDR     R1, =EFC1_FMR_Val
                STR     R1, [R0, #EFC1_FMR]
                ENDIF

; Setup WDT
                IF      WDT_SETUP != 0
                LDR     R0, =WDT_BASE
                LDR     R1, =WDT_MR_Val
                STR     R1, [R0, #WDT_MR]
                ENDIF


; Setup PMC
                IF      PMC_SETUP != 0
                LDR     R0, =PMC_BASE

;  Setup Main Oscillator
                LDR     R1, =PMC_MOR_Val
```

```
                STR     R1, [R0, #PMC_MOR]

;  Wait until Main Oscillator is stablilized
                IF      (PMC_MOR_Val:AND:PMC_MOSCEN) != 0
MOSCS_Loop      LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_MOSCS
                BEQ     MOSCS_Loop
                ENDIF

;  Setup the PLL
                IF      (PMC_PLLR_Val:AND:PMC_MUL) != 0
                LDR     R1, =PMC_PLLR_Val
                STR     R1, [R0, #PMC_PLLR]

;  Wait until PLL is stabilized
PLL_Loop        LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_LOCK
                BEQ     PLL_Loop
                ENDIF

;  Select Clock
                IF      (PMC_MCKR_Val:AND:PMC_CSS) == 1     ; Main Clock Selected
                LDR     R1, =PMC_MCKR_Val
                AND     R1, #PMC_CSS
                STR     R1, [R0, #PMC_MCKR]
WAIT_Rdy1       LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_MCKRDY
                BEQ     WAIT_Rdy1
                LDR     R1, =PMC_MCKR_Val
                STR     R1, [R0, #PMC_MCKR]
WAIT_Rdy2       LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_MCKRDY
                BEQ     WAIT_Rdy2
                ELIF    (PMC_MCKR_Val:AND:PMC_CSS) == 3     ; PLL  Clock Selected
                LDR     R1, =PMC_MCKR_Val
                AND     R1, #PMC_PRES
                STR     R1, [R0, #PMC_MCKR]
WAIT_Rdy1       LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_MCKRDY
                BEQ     WAIT_Rdy1
                LDR     R1, =PMC_MCKR_Val
                STR     R1, [R0, #PMC_MCKR]
WAIT_Rdy2       LDR     R2, [R0, #PMC_SR]
                ANDS    R2, R2, #PMC_MCKRDY
                BEQ     WAIT_Rdy2
                ENDIF   ; Select Clock
                ENDIF   ; PMC_SETUP


; Copy Exception Vectors to Internal RAM

                IF      :DEF:RAM_INTVEC
                ADR     R8, Vectors         ; Source
```

```
                LDR     R9, =RAM_BASE       ; Destination
                LDMIA   R8!, {R0-R7}        ; Load Vectors
                STMIA   R9!, {R0-R7}        ; Store Vectors
                LDMIA   R8!, {R0-R7}        ; Load Handler Addresses
                STMIA   R9!, {R0-R7}        ; Store Handler Addresses
                ENDIF


; Remap on-chip RAM to address 0

MC_BASE EQU     0xFFFFFF00      ; MC Base Address
MC_RCR  EQU     0x00            ; MC_RCR Offset

                IF      :DEF:REMAP
                LDR     R0, =MC_BASE
                MOV     R1, #1
                STR     R1, [R0, #MC_RCR]   ; Remap
                ENDIF


; Setup Stack for each mode

                LDR     R0, =Stack_Top

;  Enter Undefined Instruction Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #UND_Stack_Size

;  Enter Abort Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #ABT_Stack_Size

;  Enter FIQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #FIQ_Stack_Size

;  Enter IRQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #IRQ_Stack_Size

;  Enter Supervisor Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #SVC_Stack_Size

;  Enter User Mode and set its Stack Pointer
                ; 在跳转到__main函数前，维持在SVC模式
                ; MSR     CPSR_c, #Mode_USR
                IF      :DEF:__MICROLIB
```

```
                EXPORT __initial_sp

                ELSE

                MOV     SP, R0
                SUB     SL, SP, #USR_Stack_Size

                ENDIF

; Enter the C code

                IMPORT  __main
                LDR     R0, =__main
                BX      R0

                IMPORT rt_interrupt_enter
                IMPORT rt_interrupt_leave
                IMPORT rt_thread_switch_interrput_flag
                IMPORT rt_interrupt_from_thread
                IMPORT rt_interrupt_to_thread
                IMPORT rt_hw_trap_irq

; IRQ处理的实现
IRQ_Handler     PROC
                EXPORT IRQ_Handler
                stmfd   sp!, {r0-r12,lr}        ; 对R0 - R12，LR寄存器压栈
                bl  rt_interrupt_enter          ; 通知RT-Thread进入中断模式
                bl  rt_hw_trap_irq              ; 相应中断服务例程处理
                bl  rt_interrupt_leave          ; 通知RT-Thread要离开中断模式

                ; 判断中断中切换是否置位，如果是，进行上下文切换
                ldr r0, =rt_thread_switch_interrput_flag
                ldr r1, [r0]
                cmp r1, #1
                beq rt_hw_context_switch_interrupt_do   ; 中断中切换发生
                                                        ; 如果跳转了，将不会回来

                ldmfd   sp!, {r0-r12,lr}        ; 恢复栈
                subs    pc, lr, #4              ; 从IRQ中返回
                ENDP

; void rt_hw_context_switch_interrupt_do(rt_base_t flag)
; 中断结束后的上下文切换
rt_hw_context_switch_interrupt_do   PROC
                EXPORT rt_hw_context_switch_interrupt_do
                mov r1,  #0                     ; 清除中断中切换标志
                str r1,  [r0]

                ldmfd   sp!, {r0-r12,lr}        ; 先恢复被中断线程的上下文
                stmfd   sp!, {r0-r3}            ; 对R0 - R3压栈，因为后面会用到
                mov r1,  sp                     ; 把此处的栈值保存到R1
                add sp,  sp, #16                ; 恢复IRQ的栈，后面会跳出IRQ模式
```

```
                sub r2,  lr, #4                 ; 保存切换出线程的PC到R2

                mrs r3,  spsr                   ; 获得SPSR寄存器值
                orr r0,  r3, #I_Bit|F_Bit
                msr spsr_c, r0                  ; 关闭SPSR中的IRQ/FIQ中断

                ; 切换到SVC模式
                msr cpsr_c, #Mode_SVC

                stmfd   sp!, {r2}               ; 保存切换出任务的PC
                stmfd   sp!, {r4-r12,lr}        ; 保存R4 - R12，LR寄存器
                mov     r4,  r1                 ; R1保存有压栈R0 - R3处的栈位置
                mov     r5,  r3                 ; R3切换出线程的CPSR
                ldmfd   r4!, {r0-r3}            ; 恢复R0 - R3
                stmfd   sp!, {r0-r3}            ; R0 - R3压栈到切换出线程
                stmfd   sp!, {r5}               ; 切换出线程CPSR压栈
                mrs     r4,  spsr
                stmfd   sp!, {r4}               ; 切换出线程SPSR压栈

                ldr r4,  =rt_interrupt_from_thread
                ldr r5,  [r4]
                str sp,  [r5]                   ; 保存切换出线程的SP指针

                ldr r6,  =rt_interrupt_to_thread
                ldr r6,  [r6]
                ldr sp,  [r6]                   ; 获得切换到线程的栈

                ldmfd   sp!, {r4}               ; 恢复SPSR
                msr     SPSR_cxsf, r4
                ldmfd   sp!, {r4}               ; 恢复CPSR
                msr     CPSR_cxsf, r4

                ldmfd   sp!, {r0-r12,lr,pc}     ; 恢复R0 - R12，LR及PC寄存器
                ENDP

                IF      :DEF:__MICROLIB

                EXPORT  __heap_base
                EXPORT  __heap_limit

                ELSE
; User Initial Stack & Heap
                AREA    |.text|, CODE, READONLY

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + USR_Stack_Size)
                LDR     R2, = (Heap_Mem +      Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR
```

```
ENDIF

END
```

D.5 中断处理

中断处理部分和GNU GCC中的移植是相同的，详细请参见:ref: *AT91SAM7S64-interrupt-c*

D.6 开发板初始化

此部分代码也和GNU GCC中断移植相同，详细请参见:ref: *AT91SAM7S64-board-c*

APPENDIX

E

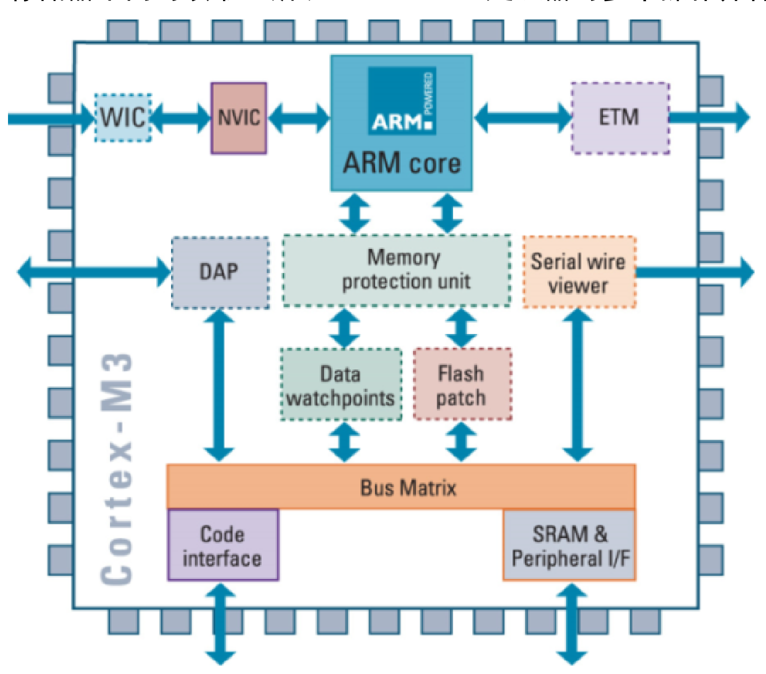
STM32说明

本附录是RT-Thread STM32移植的说明。STM32是一款ARM Cortex M3芯片，本附录也对RT-Thread关于ARM Cortex M3体系结构移植情况进行了详细说明。

E.1 ARM Cortex M3概况

Cortex M3微处理器是ARM公司于2004年推出的基于ARMv7架构的新一代微处理器，它的速度比目前广泛使用的ARM7快三分之一，功耗则低四分之三，并且能实现更小芯片面积，利于将更多功能整合在更小的芯片尺寸中。

Cortex-M3微处理器包含了一个ARM core，内置了嵌套向量中断控制器、存储器保护等系统外设。ARM core内核基于哈佛架构，3级流水线，指令和数据分别使用一条总线，由于指令和数据可以从存储器中同时读取，所以 Cortex-M3 处理器对多个操作并行执行，加快了应用程序的执行速度。



Cortex-M3 微处理器是一个 32 位处理器，包括13 个通用寄存器，两个堆栈指针，一个链接寄存器，一个程序计数器和一系列包含编程状态寄存器的特殊寄存器。Cortex-M3微处理器的指令集是Thumb-2指令，是16位Thumb指令的扩展集，可使用于多种场合。BFI 和 BFC 指令为位字段指令，在网络信息包处理等应用中可大派用场；SBFX 和 UBFX 指令改进了从寄存器插入或提取多

个位的能力，这一能力在汽车应用中的表现相当出色；RBIT 指令的作用是将一个字中的位反转，在 DFT 等 DSP 运算法则的应用中非常有用；表分支指令 TBB 和TBH用于平衡高性能和代码的紧凑性；Thumb-2指令集还引入了一个新的 If-Then结构，意味着可以有多达4个后续指令进行条件执行。

Cortex-M3 微处理器支持两种工作模式（线程模式（Thread）和处理模式（Handler））和两个等级的访问形式（有特权或无特权），在不牺牲应用程序安全的前提下实现了对复杂的开放式系统的执行。无特权代码的执行限制或拒绝对某些资源的访问，如某个指令或指定的存储器位置。Thread 是常用的工作模式，它同时支持享有特权的代码以及没有特权的代码。当异常发生时，进入 Handler模式，在该模式中所有代码都享有特权。这两种模式中分别使用不同的两个堆栈指针寄存器。

Cortex-M3微处理器的异常模型是基于堆栈方式的。当异常发生时，程序计数器、程序状态寄存器、链接寄存器和R0 - R3、R12四个通用寄存器将被压进堆栈。在数据总线对寄存器压栈的同时，指令总线从向量表中识别出异常向量，并获取异常代码的第一条指令。一旦压栈和取指完成，中断服务程序或故障处理程序就开始执行。当处理完毕后，前面压栈的寄存器自动恢复，中断了的程序也因此恢复正常的执行。由于可以在硬件中处理堆栈操作，Cortex-M3 处理器免去了在传统的 C语言中断服务程序中为了完成堆栈处理所要编写的汇编代码。

Cortex-M3微处理器内置的中断控制器支持中断嵌套（压栈），允许通过提高中断的优先级对中断进行优先处理。正在处理的中断会防止被进一步激活，直到中断服务程序完成。而中断处理过程中，它使用了tail-chaining技术来防止当前中断和和未决中断处理之间的压出栈。

E.2 ARM Cortex M3移植要点

ARM Cortex M3微处理器可以说是和ARM7TDMI微处理器完全不同的体系结构，在进行RT-Thread移植时首先需要把线程的上下文切换移植好。

通常的ARM移植，RT-Thread需要手动的保存当前模式下几乎所有寄存器，R0 – R13，LR，PC，CPSR，SPSR等。在Cortex M3微处理器中，则不需要保存全部的寄存器到栈中，因为当一个异常触发时，Cortex-M3硬件能够自动的完成部分的寄存器保存。

```
; rt_base_t rt_hw_interrupt_disable();
; 关闭中断
rt_hw_interrupt_disable    PROC
        EXPORT  rt_hw_interrupt_disable
        MRS     r0, PRIMASK                                        ; 读出PRIMASK值，即返回值
        CPSID   I                                                  ; 关闭中断
        BX      LR
        ENDP

; void rt_hw_interrupt_enable(rt_base_t level);
; 恢复中断
rt_hw_interrupt_enable    PROC
        EXPORT  rt_hw_interrupt_enable
        MSR     PRIMASK, r0                                        ; 恢复R0寄存器的值到PRIMASK中
        BX      LR
        ENDP

; void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
```

```
; r0 --> from
; r1 --> to
; 在Cortex M3移植中，这两个函数的内容都是相同的，因为正常模式的切换也采取了触发PenSV异常的方式进行
rt_hw_context_switch_interrupt
        EXPORT rt_hw_context_switch_interrupt
rt_hw_context_switch    PROC
        EXPORT rt_hw_context_switch

        ; 设置参数rt_thread_switch_interrput_flag为1，代表将要发起线程上下文切换
        LDR     r2, =rt_thread_switch_interrput_flag
        LDR     r3, [r2]
        CMP     r3, #1                              ; 参数已经置1，说明已经做过线程切换触发
        BEQ     _reswitch
        MOV     r3, #1
        STR     r3, [r2]

        LDR     r2, =rt_interrupt_from_thread      ; 保存切换出线程栈指针
        STR     r0, [r2]                            ; （切换过程中需要更新到当前位置）

_reswitch
        LDR     r2, =rt_interrupt_to_thread        ; 保存切换到线程栈指针
        STR     r1, [r2]

        LDR     r0, =NVIC_INT_CTRL
        LDR     r1, =NVIC_PENDSVSET
        STR     r1, [r0]                            ; 触发PendSV异常
        BX      LR
        ENDP

; PendSV异常处理
; r0 --> swith from thread stack
; r1 --> swith to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 等寄存器已经被自动压栈到切换出线程栈中
rt_hw_pend_sv   PROC
    EXPORT rt_hw_pend_sv

        ; 为了保护线程切换，先关闭中断
        MRS     r2, PRIMASK
        CPSID   I

        ; 获得rt_thread_switch_interrupt_flag参数，以判断pendsv是否已经处理过
        LDR     r0, =rt_thread_switch_interrput_flag
        LDR     r1, [r0]
        CBZ     r1, pendsv_exit         ; pendsv已经被处理，直接退出

        ; 清除参数：rt_thread_switch_interrput_flag为0
        MOV     r1, #0x00
        STR     r1, [r0]

        LDR     r0, =rt_interrupt_from_thread
        LDR     r1, [r0]
        CBZ     r1, swtich_to_thread    ; 如果切换出线程为0，这是第一次上下文切换
```

```
        MRS     r1, psp                 ; 获得切换出线程栈指针
        STMFD   r1!, {r4 - r11}         ; 对剩余的R4 - R11寄存器压栈
        LDR     r0, [r0]
        STR     r1, [r0]                ; 更新切换出线程栈指针

    swtich_to_thread
        LDR     r1, =rt_interrupt_to_thread
        LDR     r1, [r1]
        LDR     r1, [r1]                ; 载入切换到线程的栈指针到R1寄存器

        LDMFD   r1!, {r4 - r11}         ; 恢复R4 - R11寄存器
        MSR     psp, r1                 ; 更新程序栈指针寄存器

    pendsv_exit
        ; 恢复中断
        MSR     PRIMASK, r2

        ORR     lr, lr, #0x04           ; 构造LR以返回到Thread模式
        BX      lr                      ; 从PendSV异常中返回
        ENDP

; void rt_hw_context_switch_to(rt_uint32 to);
; r0 --> to
; 切换到函数，仅在第一次调度时调用
rt_hw_context_switch_to    PROC
        EXPORT rt_hw_context_switch_to
rt_hw_context_switch_to    PROC
        EXPORT rt_hw_context_switch_to
        LDR     r1, =rt_interrupt_to_thread             ; 设置切换到线程
        STR     r0, [r1]

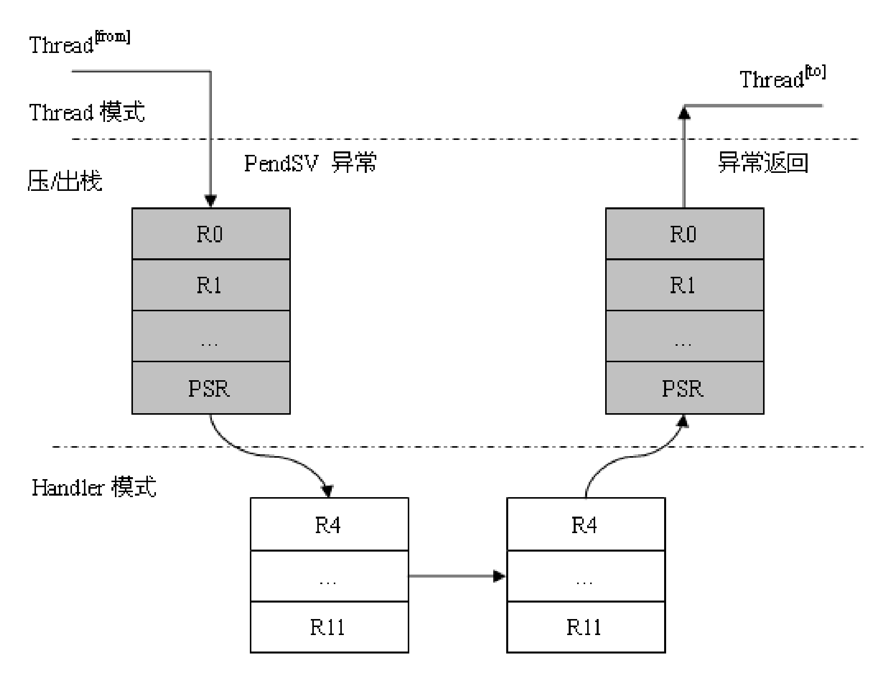
        LDR     r1, =rt_interrupt_from_thread           ; 设置切换出线程栈为0
        MOV     r0, #0x0
        STR     r0, [r1]

        LDR     r0, =NVIC_SYSPRI2                        ; 设置优先级
        LDR     r1, =NVIC_PENDSV_PRI
        STR     r1, [r0]

        LDR     r0, =NVIC_INT_CTRL
        LDR     r1, =NVIC_PENDSVSET
        STR     r1, [r0]                                 ; 触发PendSV异常

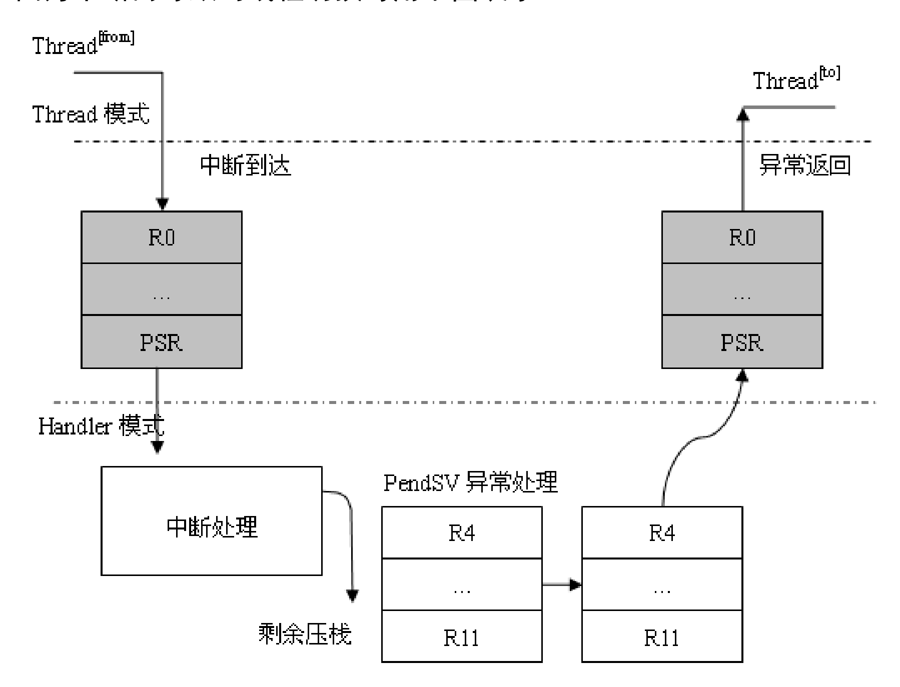
        CPSIE   I                                        ; 使能中断以使PendSV能够正常处理
        ENDP
```

正常模式下的线程上下文切换的过程可以用下图来表示：



当要进行切换时（假设从Thread [from] 切换到Thread [to]），通过rt_hw_context_switch函数触发一个PenSV异常。异常产生时，Cortex M3会把PSR，PC，LR，R0 – R3，R12自动压入当前线程的栈中，然后切换到PenSV异常处理。到PenSV异常后，Cortex M3工作模式切换到Handler模式，由函数rt_hw_pend_sv进行处理。rt_hw_pend_sv函数会载入切换出线程和切换到线程的栈指针，如果切换出线程的栈指针是0那么表示这是第一次线程上下文切换，不需要对切换出线程做压栈动作。如果切换出线程栈指针非零，则把剩余未压栈的R4 – R11寄存器依次压栈；然后从切换到线程栈中恢复R4 – R11寄存器。当从PendSV异常返回时，PSR，PC，LR，R0 – R3，R12等寄存器由Cortex M3自动恢复。

因为中断而导致的线程切换可用下图表示：



当中断达到时，当前线程会被中断并把PC，PSR，R0 – R3，R12等压到当前线程栈中，工作模式

切换到Handler模式。

在运行中断服务例程期间，如果发生了线程切换（调用rt_schedule），会先判断当前工作模式是否是Handler模式（依赖于全局变量rt_interrupt_nest），如果是则调用rt_hw_context_switch_interrupt函数进行伪切换：

> 在rt_hw_context_switch_interrupt函数中，将把当前线程栈指针赋值到rt_interrupt_from_thread变量上，把要切换过去的线程栈指针赋值到rt_interrupt_to_thread变量上，并设置中断中线程切换标志rt_thread_switch_interrput_flag为1。

在最后一个中断服务例程结束时，Cortex M3将去处理PendSV异常，因为PendSV异常的优先级是最低的，所以只有触发过PendSV异常，它将总是在最后进行处理。

E.3 RT-Thread/STM32说明

RT-Thread/STM32移植是基于RealView MDK开发环境进行移植的（GNU GCC编译器和IAR ARM编译亦支持），和STM32相关的代码大多采用RealView MDK中的代码，例如start_rvds.s是从RealView MDK自动添加的启动代码中修改而来。

和RT-Thread以往的ARM移植不一样的是，系统底层提供的rt_hw_系列函数相对要少些，建议可以考虑使用成熟的库（例如针对STM32芯片，可以采用ST官方的固件库）。RT-Thread/STM32工程中已经包含了STM32f10x系列3.1.x的库代码，可以酌情使用。

和中断相关的rt_hw_函数（RT-Thread编程指南第10章大多数函数）本移植中并不具备，所以可以跳过OS层直接操作硬件。在编写中断服务例程时，推荐使用如下的模板：

```
void rt_hw_interrupt_xx_handler(void)
{
        /* 通知RT-Thread进入中断模式 */
        rt_interrupt_enter();

        /* ... 中断处理 */

        /* 通知RT-Thread离开中断模式 */
        rt_interrupt_leave();
}
```

rt_interrupt_enter函数会通知OS进入到中断处理模式（相应的线程切换行为会有些变化）；rt_interrupt_leave函数会通知OS离开了中断处理模式。

E.4 RT-Thread/STM32移植默认配置参数

- 线程优先级支持，32优先级
- 内核对象支持命名，4字符
- 操作系统节拍单位，10毫秒
- 支持钩子函数
- 支持信号量、互斥锁

- 支持事件、邮箱、消息队列
- 支持内存池，
- 支持RT-Thread自带的动态堆内存分配器

APPENDIX

F

例程说明

RT-Thread的例子可以从RT-Thread位于 Google的svn代码版本控制服务器 中获得（examples目录中），同时也包含了部分书中完整的例程，例如线程创建等例程。

F.1 例程的基本结构

RT-Thread的例程被编写成一个个单独的文件，并且包含一个独立的用户应用入口（rt_application_init）。所以当一个例程加入到整个工程中时，只需要替换系统默认的application.c文件即可编译出这个例程处理。

一个最简单的例程就是一个包含空实现的rt_application_init：

```
/* RT-Thread的头文件，一般使用到RT-Thread的服务都需要包含这个头文件 */
#include <rtthread.h>

/* 用户代码入口函数 */
void rt_application_init()
{
    /* 空实现 */
}
```

而复杂些的例程则包括了一个属于用户自身的线程创建，例如一个线程创建的例程：

```
1  /*
2   * 程序清单：动态线程
3   *
4   * 这个程序会初始化2个动态线程，它们拥有共同的入口函数，但参数不相同
5   */
6  #include <rtthread.h>
7  #include "tc_comm.h"
8
9  /* 指向线程控制块的指针 */
10 static rt_thread_t tid1 = RT_NULL;
11 static rt_thread_t tid2 = RT_NULL;
12 /* 线程入口 */
13 static void thread_entry(void* parameter)
14 {
15         rt_uint32_t count = 0;
16         rt_uint32_t no = (rt_uint32_t) parameter; /* 获得正确的入口参数 */
```

```
17
18          while (1)
19          {
20                  /* 打印线程计数值输出 */
21                  rt_kprintf("thread%d count: %d\n", no, count ++);
22
23                  /* 休眠10个OS Tick */
24                  rt_thread_delay(10);
25          }
26  }
27
28  int thread_dynamic_simple_init()
29  {
30          /* 创建线程1 */
31          tid1 = rt_thread_create("thread",
32                  thread_entry, RT_NULL, /* 线程入口是thread1_entry, 入口参数是RT_NULL */
33                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
34          if (tid1 != RT_NULL)
35                  rt_thread_startup(tid1);
36          else
37                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
38
39          /* 创建线程2 */
40          tid2 = rt_thread_create("thread",
41                  thread_entry, RT_NULL, /* 线程入口是thread2_entry, 入口参数是RT_NULL */
42                  THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
43          if (tid2 != RT_NULL)
44                  rt_thread_startup(tid2);
45          else
46                  tc_stat(TC_STAT_END | TC_STAT_FAILED);
47
48          return 0;
49  }
50
51  #ifdef RT_USING_TC
52  static void _tc_cleanup()
53  {
54          /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
55          rt_enter_critical();
56
57          /* 删除线程 */
58          if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
59                  rt_thread_delete(tid1);
60          if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
61                  rt_thread_delete(tid2);
62
63          /* 调度器解锁 */
64          rt_exit_critical();
65
66          /* 设置TestCase状态 */
67          tc_done(TC_STAT_PASSED);
68  }
69
```

```
int _tc_thread_dynamic_simple()
{
        /* 设置TestCase清理回调函数 */
        tc_cleanup(_tc_cleanup);
        thread_dynamic_simple_init();

        /* 返回TestCase运行的最长时间 */
        return 100;
}
/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_thread_dynamic_simple, a dynamic thread example);
#else
/* 用户应用入口 */
int rt_application_init()
{
        thread_dynamic_simple_init();

        return 0;
}
#endif
```

因为RT-Thread的可裁剪性，所以有的时候并不是所有的例程都能够使用相同的配置。例如对于一个支持256个优先级的系统，当例程线程使用200优先级时，那么例程能够正常运行，但是当对于一个只支持32个优先级的系统时，200优先级的线程将直接进入ASSERT错误，系统阻止了它的运行。

为了屏蔽一些不同系统带来的差异，需要定义一些宏，这个是由tc_comm.h文件来定义的：

```
#ifndef __TC_COMM_H__
#define __TC_COMM_H__

/*
 * RT-Thread测试用例
 *
 */
#include <rtthread.h>
#include <finsh.h>

#if RT_THREAD_PRIORITY_MAX == 8
/* 当系统最大优先级是8的情况 */
#define THREAD_PRIORITY     6
#elif RT_THREAD_PRIORITY_MAX == 32
/* 当系统最大优先级是32的情况 */
#define THREAD_PRIORITY     25
#elif RT_THREAD_PRIORITY_MAX == 256
/* 当系统最大优先级是256的情况 */
#define THREAD_PRIORITY     200
#endif

/* 定义例程中用到的线程栈大小 */
#define THREAD_STACK_SIZE   512
/* 定义例程中用到的线程时间分片大小 */
#define THREAD_TIMESLICE    5
```

```
26
27 /* 针对测试用例的几个状态 */
28 #define TC_STAT_END     0x00
29 #define TC_STAT_RUNNING 0x01
30 #define TC_STAT_FAILED  0x10
31 #define TC_STAT_PASSED  0x00
32
33 #endif
```

F.2 例程向测试用例的转换

一个单独的例程是携带rt_application_init用户程序入口的独立文件，但是有的时候不可能一个个文件加入到工程进行运行，编译也需要花费很长的时间！同时做测试时也希望能够自动的进行批量的测试，而不是一个个单独地进行测试。

一个小型的测试系统应运而生：把相应的一个个例程文件都包含到一个工程中，再通过finsh shell特有的命令行系统进行单一的、重复的、多样的自动测试。

在例程中，最主要的地方是，每一个例程都会定义独立的rt_application_init用户程序入口函数。这样的话，当多个rt_application_init函数一起链接时，将会出现符号错误的问题。这个解决方法可以采用在程序中添加条件宏的方式来解决：

```
1  #ifdef RT_USING_TC
2  static void _tc_cleanup()
3  {
4      /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
5      rt_enter_critical();
6
7      /* 删除线程 */
8      if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
9          rt_thread_delete(tid1);
10     if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
11         rt_thread_delete(tid2);
12
13     /* 调度器解锁 */
14     rt_exit_critical();
15
16     /* 设置TestCase状态 */
17     tc_done(TC_STAT_PASSED);
18 }
19
20 int _tc_thread_dynamic_simple()
21 {
22     /* 设置TestCase清理回调函数 */
23     tc_cleanup(_tc_cleanup);
24     thread_dynamic_simple_init();
25
26     /* 返回TestCase运行的最长时间 */
27     return 100;
28 }
29 /* 输出函数命令到finsh shell中 */
```

```
30  FINSH_FUNCTION_EXPORT(_tc_thread_dynamic_simple, a dynamic thread example);
31  #else
32  /* 用户应用入口 */
33  int rt_application_init()
34  {
35      thread_dynamic_simple_init();
36
37      return 0;
38  }
39  #endif
```

当条件宏RT_USING_TC （意思是：使用测试用例）被定义了时，将编译第2行 - 30行的代码；而当这个宏并没被定义时，编译器将编译第32行 - 38行的代码。

而且当定义了RT_USING_TC宏时，在30行中有一条输出函数到finsh shell的语句：

```
FINSH_FUNCTION_EXPORT(_tc_thread_dynamic_simple, a dynamic thread example);
```

这条语句向finsh shell输出了_tc_thread_dynamic_simple的函数，不过很可惜的是，默认finsh shell只支持最大16个字符的变量名，所以如果完整的输入_tc_thread_dynamic_simple()函数，将不能够被finsh shell识别。

F.3 测试用例的基本结构

当需要把所有这些例程都作为测试用例来使用时（当然也包含很多其他的一些测试用例代码），如上节描述的，需要定义RT_USING_TC宏，并且需要加入tc_comm.c文件的编译。

这个这个文件中，添加了和测试相关的几条命令：

```
/* 列表当前系统中存在的测试用例 */
void list_tc(void);
/* 开始运行一个或多个测试用例，tc_prefix是测试用例的前缀 */
void tc_start(const char* tc_prefix);
/* 停止运行测试用例 */
void tc_stop(void);
```

这几个函数的解释：

- tc_start函数：

  这个函数包含一个输入参数 tc_prefix ，它指示了应该运行哪些测试用例，例如针对线程的测试包含了：

  - thread_delay – 测试线程休眠操作
  - thread_delete – 测试删除线程操作
  - …

  如果tc_prefix给出的参数是"thread"，那么将依次运行所有以_tc_thread开头的测试用例。

- tc_stop函数：

  这个函数用于停止当前正在运行的测试用例。

- list_tc函数：

    这个函数用于列表显示当前系统中存在的测试用例(自动去掉_tc_前缀进行显示)。

同时测试系统会提供以下几个API接口给测试用例：

```
/* 在测试用例中用于通知测试已经结束 */
void tc_done(rt_uint8_t state);
/* 在测试用例中设置测试过程中的状态 */
void tc_stat(rt_uint8_t state);
/* 设置一个测试用例结束时的现场清理函数 */
void tc_cleanup(void (*cleanup)(void));
```

这几个函数的解释：

- tc_done函数：

    在测试用例中用于通知测试已经结束，可以离开测试状态。是否真正结束还依赖于其他并行运行的线程情况。state参数支持如下：

    ```
    #define TC_STAT_END     0x00    /* 测试结束       */
    #define TC_STAT_RUNNING 0x01    /* 测试正在运行中 */
    #define TC_STAT_FAILED  0x10    /* 测试失败       */
    #define TC_STAT_PASSED  0x00    /* 测试通过       */
    ```

- tc_stat函数：

    在测试用例中用于设置测试过程中的测试状态。其中state参数和tc_done函数中的state意义，取值范围相同。

- tc_cleanup函数：

    测试用例用于设置一个当测试用例退出时被自动回调的函数，主要用于清除掉为测试准备的一些现场信息，例如删除线程，信号量等。

由上面几个函数基本上构成了一个简单的测试系统：

- 以_tc_为前缀名的函数输出到finsh shell，测试系统将自动依据_tc_前缀名查找系统中存在的测试用例。
- 当进行测试时，启动一个优先级稍微高一些的线程（优先级为THREAD_PRIORITY - 3），然后由这个线程调用测试里程的入口函数开始进行测试。
- 在测试用例入口函数中，根据自己测试的情况创建或初始化相应的线程，并采用tc_cleanup函数设置清理现场用的回调函数，最后返回到测试系统线程中（其返回值代表了测试系统线程最大的等待数）。
- 测试系统线程挂起在信号量上，直到测试结束或超时停止测试。
- 测试用例可以调用tc_state设置测试的状态，通过或失败；也可调用tc_done以通知给测试系统线程测试结束。